IBM External Submission #24167

Reprint Courtesy of International Business Machines Corporation, © 2015 International Business Machines Corporation

INTERNATIONAL BUSINESS MACHINES CORPORATION (IBM) ARMONK, NEW YORK 10504

PERMISSION TO REPRINT IBM COPYRIGHTED PUBLICATIONS

Each reprint must be accompanied by the following credit line: **"Reprint Courtesy of International Business Machines Corporation, © (year) International Business Machines Corporation".** The credit line normally should appear on the page where the reprint appears, either under the title or as a footnote.

If the foregoing is inconvenient, the credit line may be placed on the face or back of the title page (or front cover, if there is no title page) or in a conveniently viewable manner with suitable reference to the place where the reprint appears.

When multiple IBM materials are reprinted in the same publication, a consolidated credit paragraph may be used on the title page, or in a conveniently viewable manner listing the titles, corresponding copyright notices and references to the points where the reprints appear.

It is the understanding of **International Business Machines Corporation** that the purpose for which its material is being used is accurate and true as stated in the original request.

Permission to quote from, transmit electronically or reprint IBM material is limited to the purpose and quantities originally requested and must not be construed as a blanket license to use the material for other purposes or to reprint other IBM copyrighted material.

IBM reserves the right to withdraw permission to use copyrighted material whenever, in its discretion, it feels that the privilege of using its material is being used in a way detrimental to its interest or the above instructions are not being followed properly to protect its copyright.

No permission is granted to use trademarks of **International Business Machines Corporation** and its affiliates apart from the incidental appearance of such trademarks in the titles, text, and illustrations of the named publications. Any proposed use of trademarks apart from such incidental appearance requires separate approval in writing and ordinarily cannot be given. The use of any IBM trademark should not be of a manner which might cause confusion of origin or appear to endorse non-IBM products.

THIS PERMISSION IS PROVIDED WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

INTERNATIONAL BUSINESS MACHINES CORPORATION

Dated: August 19, 2015

Control Program Programming Guide and Reference

IBM OS/2 Toolkit Documentation

How to Use the CP Guide and Reference

The OS/2 operating system Control Program Guide and Reference is a detailed technical guide and reference for application programmers. It gives reference information and code examples to enable you to write source code using Control Program functions.

Before you begin to use this information, it would be helpful to understand how you can:

- Expand the Contents to see all available topics
- Obtain additional information for a highlighted word or phrase
- Use action bar choices
- Use the programming information.

How to Use the Contents

When the Contents window first appears, some topics have a plus (+) sign beside them. The plus sign indicates that additional topics are available.

To expand the Contents if you are using a mouse, click on the plus sign. If you are using the keyboard, use the Up or Down Arrow key to highlight the topic, and press the plus (+) key. For example, **File System** has a plus sign beside it. To see additional topics for that heading, click on the plus sign or highlight that topic and press the plus (+) key.

To view a topic, double-click on the topic (or press the Up or Down Arrow key to highlight the topic, and then press the Enter key).

How to Obtain Additional Information

After you select a topic, the information for that topic appears in a window. Highlighted words or phrases indicate that additional information is available. You will notice that certain words and phrases are highlighted in green letters, or in white letters on a black background. These are called hypertext terms. If you are using a mouse, double-click on the highlighted word. If you are using a keyboard, press the Tab key to move to the highlighted word, and then press the Enter key. Additional information then appears in a window.

How to Use Action Bar Choices

Several choices are available for managing information presented in the OS/2 operating system Control Program Guide and Reference. There are three pull-down menus on the action bar: the **Services** menu, the **Options** menu, and the **Help** menu.

The actions that are selectable from the **Services** menu operate on the active window currently displayed on the screen. These actions include the following:

Bookmark

Allows you to set a placeholder so you can retrieve information of interest to you.

When you place a bookmark on a topic, it is added to a list of bookmarks you have previously set. You can view the list, and you can remove one or all bookmarks from the list. If you have not set any bookmarks, the list is empty.

To set a bookmark, do the following:

1. Select a topic from the Contents.
2. When that topic appears, choose the **Bookmark** option from the **Services** pull-down.
3. If you want to change the name used for the bookmark, type the new name in the field.
4. Click on the **Place** radio button (or press the Up or Down Arrow key to select it).
5. Click on **OK** (or select it and press Enter). The bookmark is then added to the bookmark list.

Search

Allows you to find occurrences of a word or phrase in the current topic, selected topics, or all topics.

You can specify a word or phrase to be searched. You can also limit the search to a set of topics by first marking the topics in the Contents list.

To search for a word or phrase in all topics, do the following:

1. Choose the **Search** option from the **Services** pull-down.
2. Type the word or words to be searched for.
3. Click on **All sections** (or press the Up or Down Arrow keys to select it).
4. Click on **Search** (or select it and press Enter) to begin the search.
5. The list of topics where the word or phrase appears is displayed.

Print

Allows you to print one or more topics. You can also print a set of topics by first marking the topics in the Contents list.

To print the document Contents list, do the following:

1. Choose **Print** from the **Services** pull-down.
2. Click on **Contents** (or press the Up or Down Arrow key to select it).
3. Click on **Print** (or select it and press Enter).
4. The Contents list is printed on your printer.

Copy

Allows you to copy a topic that you are viewing to the System Clipboard or to a file that you can edit. You will find this particularly useful for copying syntax definitions and program samples into the application that you are developing.

You can copy a topic that you are viewing in two ways:

- **Copy** copies the topic that you are viewing into the System Clipboard. If you are using a Presentation Manager (PM) editor (for example, the System Editor) that copies or cuts (or both) to the System Clipboard, and pastes to the System Clipboard, you can easily add the copied information to your program source module.
- **Copy to file** copies the topic that you are viewing into a temporary file named TEXT.TMP. You can later edit that file by using any editor. You will find TEXT.TMP in the directory where your viewable document resides.

 To copy a topic, do the following:

 1. Expand the Contents list and select a topic.
 2. When the topic appears, choose **Copy to file** from the **Services** pull-down.
 3. The system puts the text pertaining to that topic into the temporary file named TEXT.TMP.

For information on one of the other choices in the **Services** pull-down, highlight the choice and press the F1 key.

The actions that are selectable from the **Options** menu allow you to change the way your Contents list is displayed. To expand the Contents and show all levels for all topics, choose **Expand all** from the **Options** pull-down. You can also press the Ctrl and * keys together. For information on one of the other choices in the **Options** pull-down, highlight the choice and press the F1 key.

The actions that are selectable from the **Help** menu allow you to select different types of help information. You can also press the F1 key for help information about the Information Presentation Facility (IPF).

How to Use the Programming Information

The Control Program Guide and Reference consists of guide and reference information that provides a detailed description of each system function.

Control Program programming information is presented by component, such as Error Management, Exception Management, and File System, for example:

```
            Contents


   + Error Management
   + Exception Management
   + File System
```

By clicking on the plus sign beside "File System", you see an alphabetic list of the Control Program functions for the file system. Selecting a function takes you directly into the reference information for that function.

Units of reference information are presented in selectable multiple windows or viewports. A viewport is a Presentation Manager window that can be sized, moved, minimized, maximized, or closed. By selecting a unit (in this case, an entry on the Contents list), you will see two windows displayed:

```
Unit Title                Selection Title

Select an item

Function Syntax
Parameters
Return Values
Notes
```

```
Example
Related Functions
Glossary
```

The window on the left is the primary window. It contains a list of items that are always available to you. The window on the right is the secondary window. It contains a 'snapshot' of the unit information. For reference units (that is, function descriptions), this window contains the Function Syntax.

All of the information needed to understand a reference unit (or topic) is readily available to you through the primary window. The information is divided into discrete information groups, and only the appropriate information group appears for the topic that you are viewing.

The information groups for a reference unit (that is, a function description) can include the following:

- Function Syntax
- Parameters
- Return Values
- Notes
- Example
- Related Functions
- Glossary

This list may vary. Some topics may be omitted when they do not apply.

Information groups are displayed in separate viewports that are stacked in a third window location that overlaps the secondary window. By selecting an item (information group) in the primary window, the item is displayed in the third window location, as follows:

```
Unit Title          Selection     Glossary

Select an item                  Select a starting
                                letter of
   .                            glossary terms
   .
   .                            A    N
   .                            B    O
   .                            C    P
Glossary                        .    .
                                .    .
                                .    .
                                M    Z
```

By selecting successive items from the primary window, additional windows are displayed on top of the previous windows displayed in the third window location. For example, in a function description, Parameters and Return Values are items listed in the primary window. When selected, they appear one on top of the other in the third window location. Because of this, you may move the first selected (topmost) window to the left before selecting the next item. This allows simultaneous display of two related pieces of information from the "stack" of windows in the third window location, as follows:

```
Unit Title         Parameters     Return Values

Select an item
   .
   .
   .
Parameters
Return Values
   .
   .
   .
```

Each window can be individually closed from its system menu. All windows are closed when you close the primary window.

Some secondary windows may have the appearance of a split screen. For example, an illustration may appear in the left half of the window, and scrollable, explanatory information may appear in the right half of the window. Because illustrations may not necessarily fit into the small window size on your screen, you may maximize the secondary window for better readability.

Introduction to the Control Program

The purpose of this reference is to give information about functions, constants, and data structures. It provides information about the functions which enable the user to call functions in the C programming language.

The following information is provided:

- The syntax and parameters for each function
- The syntax of each data type and structure

Control Program Fundamentals

OS/2* is an advanced, multitasking, operating system for personal computers. OS/2 has a rich application programming interface (API) that supports:

- Multitasking
- Interprocess communication
- Error and exception handling
- Dynamic (run-time) linking
- Graphical user interface (GUI)

OS/2 also provides advanced file system features such as installable file systems, extended file attributes, and long file names.

OS/2 runs on and takes full advantage of the 80386 microprocessor. Features based on the 80386 microprocessor include:

- 32-bit operands and operations
- Flat (non-segmented) 4 gigabyte address space
- 32-bit memory pointers
- Hardware-based memory protection and task management

OS/2 provides a windowed, graphical user interface called the Presentation Manager* (PM*).

Control Program Functionality

The lowest-level functions supplied by the OS/2 operating system are those provided by the kernel and the kernel's subsystems-the control programs of the operating system. The Control Program functions involve the most basic aspects of program execution, such as memory management, file handling, and process, thread, and session management. They also involve more sophisticated programming tasks, such as exception handling and interprocess communications. The names of all the system functions in the Control Program API are prefixed with the letters "Dos", as in DosAllocMem.

This book describes the following topics from the operating system's Control Program API:

- File and disk management
- Memory management
- Program execution control (Process and session management)
- Semaphores
- Pipes
- Queues
- Timing functions
- Error handling
- Exception handling
- Device I/O support
- Message management
- National language support and code page management
- Debugging
- Dynamic Linking

File Systems and File Management

The file system is the component of the operating system that supports storing information on mass storage devices, such as hard disks and

diskettes. Applications view a file as a logical sequence of data; the file system manages the physical locations of data on the storage device for the application and specifies how the device and the stored information are formatted.

The file system also manages file I/O operations. Applications use file system functions to open, read, write, and close disk files. File system functions enable an application to maintain the disk that holds the files-volumes, directories, and files on the disks of the computer. Applications also use OS/2 file system functions to perform I/O operations to pipes and peripheral devices connected to the computer, such as the printer.

The file system also supports redirection of input and output, for example redirecting output from the display to a disk file or to the printer.

There are two types of file systems supported by the OS/2 operating system. The first is the File Allocation Table (FAT) file system. The FAT file system is the file system used by DOS The second type of file system supported by the operating system is the installable file system (IFS). Installable file systems are external to the base operating system and are loaded by the operating system when the computer is started. The High Performance File System (HPFS) included with the OS/2 operating system is an installable file system.

Memory Management

The key features of OS/2 memory management are paged virtual memory and a 32-bit linear (flat) address space that is mapped through page tables to physical memory. This is in contrast to the segmented memory model used in previous versions of the operating system.

An application can allocate memory for its own use or to be shared with other applications.

Program Execution and Control

Multitasking is the ability of the operating system to manage the execution of more than one application at a time. For the programmer, this includes the ability to multitask your own programs.

The OS/2 operating system supports two forms of multitasking for programs. An application can start other programs that will execute concurrently with the application. These programs can be a new copy of the application, a related program that is designed to work with the application, or an unrelated program. The operating system provides functions to communicate with and control the programs started by the application.

The OS/2 operating system also enables applications to run multiple threads of execution within the same application; separate activities can be multitasked within the application. An example of this is dispatching a separate subroutine to load a file and having the subroutine execute at the same time the main routine continues to monitor and respond to user input.

Semaphores

Semaphores signal the beginning and ending of an operation and provide mutually exclusive ownership of resources. Typically, semaphores are used to prevent more than one process or thread within a process from accessing a resource, such as shared memory, at the same time.

Semaphores are defined by the system and reside in an internal memory buffer. They are divided into three types, according to the functionality they provide:

- Event semaphores enable a thread to notify waiting threads that an event has occurred.
- Mutual exclusion (mutex) semaphores enable threads to serialize their access to shared resources.
- Multiple Wait (muxwait) semaphores enable threads to wait either for multiple events to occur, or for multiple resources to become available.

Pipes

A pipe is a named or unnamed buffer used to pass data between processes. A process writes to or reads from a pipe as though the pipe were

standard input or standard output. A parent process can use pipes to control the input that a child process receives and to receive the output that the child process produces. There are two types of pipes-named and unnamed.

Queues

A queue is a named, ordered list of elements that is used to pass information between related or unrelated processes. The owner of the queue (the server process) can choose the order in which to read incoming information and can examine queue elements without removing them from the queue. Queue elements can be added and accessed in First-In-First-Out (FIFO), Last-In-First-Out (LIFO), or priority-based order.

Timers

Timers enable an application to suspend operation for a specific length of time, to block a thread of execution until an interval has elapsed, or to post an event semaphore at repeated intervals.

Error Management

The OS/2 operating system provides functions to facilitate error processing. DosErrClass returns a classification of the error and a recommended action. DosError enables an application to prevent the operating system from displaying the default error message in a pop-up window when either a hard error or a software exception occurs.

Exception Management

A multitasking operating system must manage applications carefully. A serious error (such as an attempt to access protected memory) occurring in one application cannot be permitted to damage any other application in the system. To manage errors that might damage other applications, the OS/2 operating system defines a class of error conditions called *exceptions* and defines default actions for those errors.

An exception is an abnormal condition that can occur during program execution. Common causes of exceptions include I/O errors and access protection violations. When an exception is caused by the user pressing Ctrl+Break or Ctrl+C, the exception is called a signal exception.

When an exception occurs, the default action usually taken by the operating system is to terminate the application that caused the exception. An application can register its own exception handling routine and try to handle the exception itself. It might be possible for the application to correct the condition that caused the exception and continue execution.

Device I/O

A device is a piece of hardware used for input and output. Devices used with computers include the keyboard, video display, mouse, diskette and hard disk drives, and external systems, such as modems and printers. The operating system supplies functions that can be used to access and control such devices.

Message Management

For full-screen applications, text messages-used by an application to display information to the user-can be held in a *message file* . Keeping an application's messages in a message file simplifies changing those messages, which, for example, can facilitate marketing an application in several countries simultaneously.

National Language Support and Code Page Management

Many applications must support more than one national language, for example, French and German. This requirement is simplified through the use of such resources as string tables, menu templates, dialog templates, and accelerator tables.

A *code page* is a table that defines how characters are encoded. Code page management enables a user to select a code page for keyboard input, and screen and printer output before starting an application, a system command, or a utility program in the OS/2 multitasking environment.

Debugging

Debugging is the process of detecting, diagnosing, and eliminating errors in programs. A debugger program is designed to interact with the application that it is debugging. Because of the protected mode architecture of the OS/2 operating system, special steps must be taken to allow a debugging program to perform its functions in the program being debugged. DosDebug enables one application to control the execution of another application for debugging purposes.

I/O Control Functions

The OS/2 operating system provides functions to allow an application or subsystem to send device-specific control commands to a physical device driver. These IOCtls are subfunctions issued through the DosDevIOCtl function.

For a complete listing of IOCtl functions, see Generic IOCtl Commands.

Dynamic Linking

Dynamic linking permits several applications to use a single copy of an executable module. The executable module is completely separate from the applications that use it. Several functions can be built into a DLL, and applications can use these functions as though they were part of the application's executable code. You can change the dynamically-linked functions without recompiling or relinking the application.

Notation Conventions

The following notation conventions are used in this reference:

NULL	The term NULL applied to a parameter is used to indicate the presence of the pointer parameter, but with no value.
NULLHANDLE	The term NULLHANDLE applied to a parameter is used to indicate the presence of the handle parameter, but with no value.
Implicit Pointer	If no entry for a data type "Pxxxxxxx" is found in Data Types, then it is implicitly a pointer to the data type "xxxxxxx". See Implicit Pointer Data Types for more information about implicit pointers.
Constant Names	All constants are written in uppercase to match the header files. Where applicable, constant names have a prefix derived from the name of a function, message, or idea associated with the constant. For example:

WM_CREATE	Window message
SV_CXICON	System value
CF_TEXT	Clipboard format.

In this book, references to a complete set of constants with a given prefix is written as shown in the following examples:

Window message	WM_*
System value	SV_*

Parameters and Fields — Function parameters and data structure fields are shown in italics.

Conventions Used in Function Descriptions

This section provides information about the notation conventions and function descriptions used in this reference.

The documentation for each function contains the following sections:

Syntax

The function syntax describes the C-language calling syntax of the function and gives a brief description.

Programming Note

The functions in this book are spelled in mixed-case for readability but are known to the system as uppercase character strings. If you are using a compiler that generates a mixed-case external name, you should code the functions in uppercase.

Parameters

Each parameter is listed with its C-language data type, parameter type, and a brief description.

- All data types are written in uppercase to match the header files. A data type of "Pxxxxxxx" implicitly defines a pointer to the data type "xxxxxxx".

 The term NULL applied to a parameter indicates the presence of the parameter, with no value.

 Refer to Data Types for a complete list of all data types and their descriptions.

- There are three parameter types:

Input	Specified by the programmer.
Output	Returned by the function.
Input/Output	Specified by the programmer and modified by the function.

- A brief description is provided with each parameter. Where appropriate, restrictions are also included. In some cases, the parameter points to a structure.

Returns

A list of possible return codes or errors (when appropriate) is included in this section. Some functions do not have return codes. Refer to Errors for a complete list of all return codes and their descriptions.

Remarks

This section contains additional information about the function, when required.

Related Functions

This list shows the functions (if any) that are related to the function being described.

Example Code

An example for each function is shown in the C programming language.

Error Severities

Each of the error conditions given in the list of errors for each function falls into one of these areas:

Warning

The function detected a problem, but took some remedial action that enabled the function to complete successfully. The return code in this case indicates that the function completed successfully.

Error

The function detected a problem for which it could not take any sensible remedial action. The system has recovered from the problem, and the state of the system, with respect to the application, remains the same as at the time when the function was requested. The system has not even partially executed the function (other than reporting the error).

Severe Error
: The function detected a problem from which the system could not reestablish its state, with respect to the application, at the time when that function was requested; that is, the system partially executed the function. This necessitates the application performing some corrective activity to restore the system to some known state.

Unrecoverable Error
: The function detected some problem from which the system could not re-establish its state, with respect to the application, at the time when that call was issued. It is possible that the application cannot perform some corrective action to restore the system to some known state.

Header Files

All functions require an "#include" statement for the system header file OS2.H:

```
#include  <OS2.H>
```

Most functions also require a "#define" statement to select an appropriate (conditional) section of the header file, and hence, the required prototype. Where this is necessary, it is shown at the head of the function definition in the form:

```
#define   INCL_name
```

Note: These "#define" statements must precede the "#include <OS2.H>" statement.

Addressing Elements in Arrays

Constants defining array elements are given values that are zero-based in C; that is, the numbering of the array elements starts at zero, not one.

For example, in the DevQueryCaps function, the sixth element of the *ulArray* parameter is CAPS_HEIGHT, which is equated to 5.

Count parameters related to such arrays always mean the actual number of elements available; therefore, again using the DevQueryCaps function as an example, if all elements up to and including CAPS_HEIGHT are provided for, *lCount* could be set to (CAPS_HEIGHT+1).

In functions for which the starting array element can be specified, this is always zero-based, and so the C element number constants can be used directly. For example, to start with the CAPS_HEIGHT element, the *lStart* parameter can be set to CAPS_HEIGHT.

Implicit Pointer Data Types

A data type name beginning with "P" (for example, PERRORCODE) is likely to be a pointer to another data type (in this instance, ERRORCODE).

In the data type summary, Data Types, no explicit "typedefs" are shown for pointers; therefore, if no data type definition can be found in the summary for a data type name "Pxxxxxx", it represents a pointer to the data type "xxxxxx", for which a definition should be found in the reference.

The implicit type definition needed for such a pointer "Pxxxxxx" is:

```
typedef xxxxxx *Pxxxxxx;
```

Such definitions are provided in the header files.

Storage Mapping of Data Types

The storage mapping of the data types is dependent on the machine architecture. To be portable, applications must access the data types using the definitions supplied for the environment in which they will execute.

Double-Byte Character Set (DBCS)

Throughout this publication, you will see references to specific value for character strings. The values are for single-byte character set (SBCS). If you use the double-byte character set (DBCS), note that one DBCS character equals two SBCS characters.

Programming Considerations

This section provides information you need to consider before you begin programming with Control Program functions.

Stack Size

Existing 16-bit applications (small and tiny models) must have a 4KB stack available when they enter system calls; otherwise, the stack can overflow into the data area.

Presentation Manager

The Presentation Manager component of the OS/2* operating system is based on the IBM Systems Application Architecture* (SAA*) Common Programming Interface-an architecture for the design and development of applications.

The Presentation Manager component implements the Common User Access* (CUA*) interface, which you can use to attain consistency in the appearance and behavior of your applications.

Addressing Capabilities

The operating system supports the addressing capabilities of the Intel** 80386, 80386SX, 80387, 80387 NPX, 80486, and Pentium** processors, with page-level memory protection. One page is 4KB (KB equals 1024 bytes) of contiguous physical memory.

C++ Considerations

This section contains several topics you should take into consideration if you are using C++.

C++ Header Files

OS/2 functions that used to take a PSZ as a parameter, and that do not modify the contents of the passed string, have been updated in the C++ header files to take a PCSZ data type parameter. The use of PCSZ allows for better optimization by the compiler and is more semantically compatible with C++. Existing code that calls functions that use PSZ will continue to work correctly.

Several of the typedefs have been changed in the C++ header files. For example, many items that are unsigned char in the C header files are char in the C++ header files. For instance,

```
typedef unsigned char BYTE;
```

has changed to

```
typedef char BYTE;
```

The existing samples that are included in the IBM Developer's Toolkit for OS/2 Warp Version 3 can be used with either set of the header files.

PCSZ Data Type

Note: The PCSZ data type is defined in the C++ header files included with this product. The use of the "const" keyword is not necessarily specific to C++. Certain C compilers support it as well.

If a function takes as a parameter a string that is not changed by the function, the string parameter can be declared as a "const" string or a PCSZ. PCSZ is defined in the C++ header files as a "const" pointer to a NULL-delimited string. The "const" indicates that the function will not change the contents of the string.

Declaring the parameter as PCSZ informs the C++ compiler that the function will not change the string. Therefore, the compiler simply passes a pointer to the string in the function parameter list. If the parameter is declared as a normal PSZ (not "const"), the compiler assumes that the function might change the string. Under these circumstances the compiler will add code to make a copy of the string and then pass a pointer to the copy, rather than pass a pointer to the original string.

A smaller, faster executable is often produced if the data item passed in a parameter list is declared as "const".

If the data item is declared as "const" then it must not be changed by the function.

LINK386

The C++ compiler will provide a dynamic link library which is used by LINK386 when generating error messages. This DLL will convert a compiler generated mangled name into the function prototype. If the DLL is not present, an error message will be displayed and LINK386 will display the compiler-generated, mangled name in the error message.

Control Program Functions

This chapter contains an alphabetic list of the Control Program functions available in the OS/2 operating system. These functions can be used in full-screen, windowed (VIO), and Presentation Manager sessions to perform basic operating-system functions, such as file input and output, memory allocation, thread control and communication, process control and communication, pipe and queue manipulation, and exception and error handling.

The following table lists the available functions.

Date and Time

DosGetDateTime
DosSetDateTime

Debugging

DosDebug

Device Handling

DosCloseVDD
DosOpenVDD
DosDevConfig
DosRequestVDD
DosDevIOCtl
DosTmrQueryFreq
DosTmrQueryTime

Dynamic Link Modules

DosFreeModule
DosQueryModuleName
DosLoadModule
DosQueryProcAddr
DosQueryModuleHandle
DosQueryProcType

Error Processing

DosErrClass
DosError

Exceptions

DosAcknowledgeSignalException
DosSetExceptionHandler
DosEnterMustComplete
DosSetSignalExceptionFocus
DosExitMustComplete
DosUnsetExceptionHandler
DosRaiseException
DosUnwindException
DosSendSignalException

Extended Attributes

DosEnumAttribute
DosProtectEnumAttribute

File Management - File Locking

DosCancelLockRequest
DosProtectSetFHState
DosProtectClose
DosProtectSetFileInfo
DosProtectOpen
DosProtectSetFileLocks
DosProtectQueryFileInfo
DosProtectSetFilePtr
DosProtectRead
DosProtectSetFileSize
DosProtectQueryFHState
DosProtectWrite

File Management

General

DosClose
DosFindNext
DosCopy
DosForceDelete
DosCreateDir
DosMove
DosDeleteDir
DosOpen
DosDelete
DosRead
DosDupHandle
DosSearchPath
DosEditName
DosSetMaxFH
DosFindClose
DosSetVerify
DosFindFirst
DosWrite

Queries

DosQueryAppType
DosQueryCurrentDir
DosQueryCurrentDisk
DosQueryFHState
DosQueryFileInfo
DosQueryPathInfo
DosQueryHType
DosQueryVerify

Settings

DosResetBuffer
DosSetCurrentDir
DosSetDefaultDisk
DosSetFHState
DosSetFileInfo
DosSetFileLocks
DosSetFilePtr
DosSetFileSize
DosSetPathInfo
DosSetRelMaxFH

File Systems

DosFSAttach
DosFSCtl
DosPhysicalDisk
DosQueryFSAttach
DosQueryFSInfo
DosSetFSInfo

General

DosBeep
DosShutdown

Memory Management

DosAllocMem
DosAllocSharedMem
DosAllocThreadLocalMemory
DosFreeMem
DosFreeThreadLocalMemory
DosGetNamedSharedMem
DosGetSharedMem
DosGiveSharedMem
DosQueryMem
DosSetMem
DosSubAllocMem
DosSubFreeMem
DosSubSetMem
DosSubUnsetMem

Message Management

DosGetMessage
DosInsertMessage
DosPutMessage
DosQueryMessageCP

National Language Support

DosMapCase
DosQueryCollate
DosQueryCp
DosQueryCtryInfo
DosQueryDBCSEnv
DosSetProcessCp

Pipes

DosCallNPipe
DosConnectNPipe
DosCreatePipe
DosCreateNPipe
DosDisConnectNPipe
DosPeekNPipe
DosQueryNPipeInfo
DosQueryNPipeSemState
DosQueryNPHState
DosSetNPHState
DosSetNPipeSem
DosTransactNPipe
DosWaitNPipe

Processes, Threads and Sessions

DosCreateThread
DosSelectSession

DosEnterCritSec
DosExecPgm
DosExitCritSec
DosExit
DosExitList
DosGetInfoBlocks
DosKillProcess
DosKillThread
DosResumeThread
DosSetPriority
DosSetSession
DosSleep
DosStartSession
DosStopSession
DosSuspendThread
DosWaitChild
DosWaitThread
DosQueryThreadContext

Query System Information

DosQuerySysInfo
DosScanEnv

Queues

DosCloseQueue
DosCreateQueue
DosOpenQueue
DosPeekQueue
DosPurgeQueue
DosQueryQueue
DosReadQueue
DosWriteQueue

Resource Objects

DosFreeResource
DosGetResource
DosQueryResourceSize

Semaphores

Event Semaphores

DosCloseEventSem
DosCreateEventSem
DosOpenEventSem
DosPostEventSem
DosQueryEventSem
DosResetEventSem
DosWaitEventSem

Mutex Semaphores

DosCloseMutexSem
DosCreateMutexSem
DosOpenMutexSem
DosQueryMutexSem
DosReleaseMutexSem
DosRequestMutexSem

Muxwait Semaphores

DosAddMuxWaitSem
DosCloseMuxWaitSem
DosCreateMuxWaitSem
DosDeleteMuxWaitSem
DosOpenMuxWaitSem
DosQueryMuxWaitSem
DosWaitMuxWaitSem

Timers

DosAsyncTimer
DosStartTimer
DosStopTimer

DosAcknowledgeSignalException

DosAcknowledgeSignalException - Syntax

Indicates that a process wants to receive further signals.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

ULONG     ulSignalNum;  /*  The number of the signal to be acknowledged. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosAcknowledgeSignalException(ulSignalNum);
```

DosAcknowledgeSignalException Parameter - ulSignalNum

ulSignalNum (ULONG) - input
The number of the signal to be acknowledged.

Valid signals are shown in the following list:

1	XCPT_SIGNAL_INTR
3	XCPT_SIGNAL_KILLPROC
4	XCPT_SIGNAL_BREAK

DosAcknowledgeSignalException Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosAcknowledgeSignalException returns one of the following values:

0	NO_ERROR
209	ERROR_INVALID_SIGNAL_NUMBER

For a full list of error codes, see Errors.

DosAcknowledgeSignalException - Parameters

ulSignalNum (ULONG) - input
The number of the signal to be acknowledged.

Valid signals are shown in the following list:

1	XCPT_SIGNAL_INTR
3	XCPT_SIGNAL_KILLPROC

4	XCPT_SIGNAL_BREAK

ulrc (APIRET) - returns
Return Code.

DosAcknowledgeSignalException returns one of the following values:

0	NO_ERROR
209	ERROR_INVALID_SIGNAL_NUMBER

For a full list of error codes, see Errors.

DosAcknowledgeSignalException - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosAcknowledgeSignalException is used to tell the system that the process wishes to receive further signal exceptions.

This function may be used by any thread in the process, but will only affect thread 1.

See System Exceptions for a detailed list of the system exceptions.

DosAcknowledgeSignalException - Related Functions

Related Functions

- DosEnterMustComplete
- DosExitMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler
- DosUnwindException

DosAcknowledgeSignalException - Example Code

This program demonstrate how to acknowledge a signal exception.

```
#define INCL_DOSPROCESS       /* DOS process values (for DosSleep) */
#define INCL_DOSEXCEPTIONS    /* DOS exception values */
#define INCL_ERRORS           /* DOS error values     */
#include <os2.h>
#include <stdio.h>

ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv );

int main (VOID)
{
  EXCEPTIONREGISTRATIONRECORD  RegRec  = {0};
  APIRET                       rc      = NO_ERROR;   /* Return code */

  /* Add MyTermHandler to this thread's chain of exception handlers */

  RegRec.ExceptionHandler = (ERR)MyTermHandler;
  rc = DosSetExceptionHandler( &RegRec );
```

```
  if (rc != NO_ERROR) {
     printf("DosSetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Stop this program using Ctrl-C or Ctrl-Break.\n");

  rc = DosSleep(10000L);   /* Give user 10 seconds... */

  rc = DosUnsetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosUnsetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Program ends.\n");
  return NO_ERROR;
}

/*********************************************************************/
ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv )
{
  APIRET   rc  = NO_ERROR;   /* Return code */

  printf("*** MyTermHandler entered ***\n");

  if (p1->ExceptionNum == XCPT_SIGNAL) {
        rc = DosAcknowledgeSignalException ( XCPT_SIGNAL_BREAK );
        if (rc != NO_ERROR) {
          printf("DosAcknowledgeSignalException error:
                  return code = %u\n", rc);
          return 1;
        } else {
          printf("DosAcknowledgeSignalException done. Program resumes.\n");
          return XCPT_CONTINUE_EXECUTION;  /* Continue execution */
        } /* endif */
  }

  return XCPT_CONTINUE_SEARCH;       /* Exception not resolved. */
}
```

DosAcknowledgeSignalException - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosAddMuxWaitSem

DosAddMuxWaitSem - Syntax

Adds a mutex semaphore or an event semaphore to a muxwait-semaphore list.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMUX          hmux;     /*  The handle of the muxwait semaphore that is to receive the additional semaphore. */
PSEMRECORD    pSemRec;  /*  A pointer to the semaphore record that is to be added to the muxwait list. */
APIRET        ulrc;     /*  Return Code. */

ulrc = DosAddMuxWaitSem(hmux, pSemRec);
```

DosAddMuxWaitSem Parameter - hmux

hmux (HMUX) - input
The handle of the muxwait semaphore that is to receive the additional semaphore.

DosAddMuxWaitSem Parameter - pSemRec

pSemRec (PSEMRECORD) - input
A pointer to the semaphore record that is to be added to the muxwait list.

DosAddMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosAddMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
100	ERROR_TOO_MANY_SEMAPHORES
105	ERROR_SEM_OWNER_DIED
284	ERROR_DUPLICATE_HANDLE
292	ERROR_WRONG_TYPE

For a full list of error codes, see Errors.

DosAddMuxWaitSem - Parameters

hmux (HMUX) - input
The handle of the muxwait semaphore that is to receive the additional semaphore.

pSemRec (PSEMRECORD) - input
A pointer to the semaphore record that is to be added to the muxwait list.

ulrc (APIRET) - returns
Return Code.

DosAddMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
100	ERROR_TOO_MANY_SEMAPHORES
105	ERROR_SEM_OWNER_DIED
284	ERROR_DUPLICATE_HANDLE
292	ERROR_WRONG_TYPE

For a full list of error codes, see Errors.

DosAddMuxWaitSem - Related Functions

Related Functions

- DosCloseMutexSem
- DosCreateMuxWaitSem
- DosDeleteMuxWaitSem
- DosOpenMuxWaitSem
- DosQueryMuxWaitSem
- DosWaitMuxWaitSem

DosAddMuxWaitSem - Example Code

This example creates two semaphore record lists, with two event semaphores in each. It then creates a muxwait semaphore with the first semaphore record list, and it adds the second list later using DosAddMuxWaitSem.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX      hmuxHandAny = NULLHANDLE;       /* Muxwaithandle */
 HEV       hevA[2]      = {0};             /* Event semaphores */
 SEMRECORD apsrA[2]     = {{0}};           /* Semaphore records */
 HEV       hevB[2]      = {0};             /* Event semaphores */
 SEMRECORD apsrB[2]     = {{0}};           /* Semaphore records */
 APIRET    rc          = NO_ERROR;         /* Return code */
 ULONG     ulLoop      = 0;               /* Loop count */

for (ulLoop = 0; ulLoop < 2; ulLoop++) {
    rc = DosCreateEventSem((PSZ) NULL,
                           &hevA[ulLoop],
                           0,
                           FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
    apsrA[ulLoop].hsemCur = (HSEM) hevA[ulLoop],
    apsrA[ulLoop].ulUser = 0;

    rc = DosCreateEventSem((PSZ) NULL,
                           &hevB[ulLoop],
                           0,
                           FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
```

```
    apsrB[ulLoop].hsemCur = (HSEM) hevB[ulLoop],
    apsrB[ulLoop].ulUser = 0;
} /* endfor */

rc = DosCreateMuxWaitSem((PSZ) NULL,
                         &hmuxHandAny,
                         2L,             /* Number of semaphores in list */
                         apsrA,          /* Semaphore list */
                         DCMW_WAIT_ANY); /* Wait for any semaphore */
    if (rc != NO_ERROR) {
      printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosAddMuxWaitSem(hmuxHandAny,  apsrB);
    if (rc != NO_ERROR) {
      printf("DosAddMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

return NO_ERROR;
}
```

DosAddMuxWaitSem - Topics

Select an item:

Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosAllocMem

DosAllocMem - Syntax

Allocates a private memory object within the virtual-address space.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PPVOID    ppb;   /*  A pointer to a variable that will receive the base address of the allocated private memory
ULONG     cb;    /*  Size, in bytes, of the private memory object to allocate. */
ULONG     flag;  /*  Allocate attribute and desired access protection flags. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosAllocMem(ppb, cb, flag);
```

DosAllocMem Parameter - ppb

ppb (PPVOID) - output
A pointer to a variable that will receive the base address of the allocated private memory object.

DosAllocMem Parameter - cb

cb (ULONG) - input
Size, in bytes, of the private memory object to allocate.

The size is rounded up to the next page-size boundary. The size of a page is 4KB.

DosAllocMem Parameter - flag

flag (ULONG) - input
Allocate attribute and desired access protection flags.

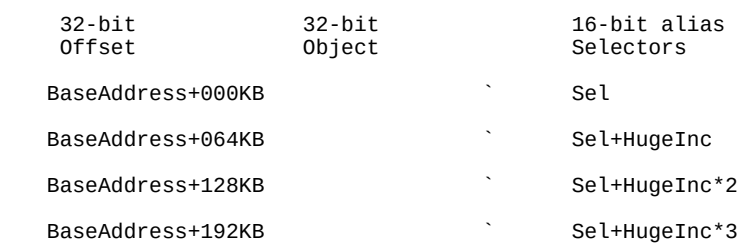
A set of flags describing the allocation attributes and desired access protection for the private memory object. Possible values are shown in the lists below:

Allocation Attributes

PAG_COMMIT (0x00000010)
All pages in the private memory object are initially committed.

OBJ_TILE (0x00000040)
The private memory object must be allocated in the first 512MB of virtual-address space, with 16-bit selectors mapping the memory object.

The 16-bit selectors are allocated to map the 32-bit object at 64KB boundaries. The figure below shows how the 16-bit alias selectors map the 32-bit object.



HugeInc is the huge increment used for DosAllocHuge.

Desired Access Protection

PAG_EXECUTE (0x00000004)
Execute access to the committed pages in the private memory object

PAG_READ (0x00000001)
Read access

PAG_WRITE (0x00000002)
Write access

PAG_GUARD (0x00000008)
Access to the committed pages in the private memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set.

DosAllocMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosAllocMem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT

For a full list of error codes, see Errors.

DosAllocMem - Parameters

ppb (PPVOID) - output
A pointer to a variable that will receive the base address of the allocated private memory object.

cb (ULONG) - input
Size, in bytes, of the private memory object to allocate.

The size is rounded up to the next page-size boundary. The size of a page is 4KB.

flag (ULONG) - input
Allocate attribute and desired access protection flags.

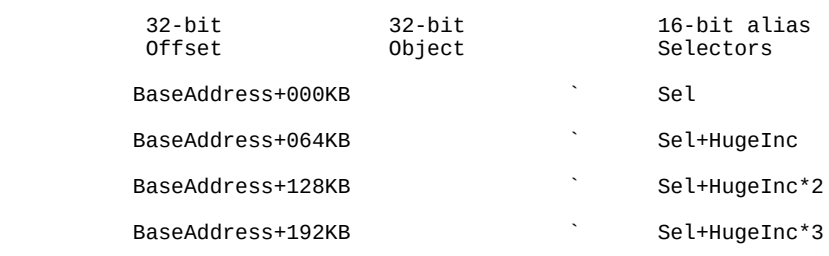
A set of flags describing the allocation attributes and desired access protection for the private memory object. Possible values are shown in the lists below:

Allocation Attributes

PAG_COMMIT (0x00000010)
All pages in the private memory object are initially committed.

OBJ_TILE (0x00000040)
The private memory object must be allocated in the first 512MB of virtual-address space, with 16-bit selectors mapping the memory object.

The 16-bit selectors are allocated to map the 32-bit object at 64KB boundaries. The figure below shows how the 16-bit alias selectors map the 32-bit object.



HugeInc is the huge increment used for DosAllocHuge.

Desired Access Protection

PAG_EXECUTE (0x00000004)
Execute access to the committed pages in the private memory object

PAG_READ (0x00000001)
Read access

PAG_WRITE (0x00000002)
Write access

PAG_GUARD (0x00000008)
Access to the committed pages in the private memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set.

ulrc (APIRET) - returns
Return Code.

DosAllocMem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT

For a full list of error codes, see Errors.

DosAllocMem - Remarks

DosAllocMem can be used to reserve, or reserve and commit, linear address space for a private memory object.

The operating system allocates a range of private pages large enough to fulfill the specified allocation request from the private virtual-address space of the subject process. The base address of the object is returned in the *ppb* parameter.

The allocated memory object is rounded up to a multiple of 4KB in size.

The committed memory allocated by DosAllocMem can be swapped.

Any access protection can be applied to committed pages within a private memory object. Committed pages are initially allocated and backed by demand pages. The first attempt to read or write the page causes a page of zeros to be created.

If a failure occurs during the allocation, no pages are allocated, and an appropriate error code is returned.

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

The guard-page attribute is intended to provide automatic stack-growth and stack-limit checking. An application may also use it in other data structures.

Reserved pages that are not committed are given an access protection of "no access".

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded up to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosAllocMem - Related Functions

Related Functions

- DosAllocSharedMem
- DosFreeMem

DosAllocMem - Example Code

This example allocates memory as Read/Write, queries it, commits it, uses it, and then releases it.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main (VOID)
{
   PVOID  MyObject    = NULL;     /* Pointer to memory object           */
   ULONG  ulObjSize   = 0;        /* Size of memory object (in bytes)   */
   ULONG  ulMemFlags  = 0;        /* Attribute flags for the object     */
   ULONG  ulMemSize   = 0;        /* Size of memory region for DosSetMem */
   APIRET rc          = NO_ERROR; /* Return code                        */

   ulObjSize = 2000;          /* Will be rounded to a page boundary - 4096 */

   rc = DosAllocMem(&MyObject,     /* Pointer to memory object pointer     */
                    ulObjSize,     /* Size of object to be allocated       */
                    PAG_WRITE );   /* Allocate memory as read/writeable    */

   if (rc != NO_ERROR) {
      printf("DosAllocMem error:  return code = %u\n",rc);
      return 1;
   }
       /* Object can't be used until it is COMMITTED.  Since this was
          not done at DosAllocMem time, do it now.                    */

   rc = DosSetMem(MyObject,                    /* Pointer to object      */
                  ulObjSize,                   /* Size of area to change */
                  PAG_DEFAULT | PAG_COMMIT );  /* Commit the object      */

   if (rc != NO_ERROR) {
      printf("DosSetMem error:  return code = %u\n",rc);
      rc = DosFreeMem(MyObject);
      /* If omitted, OS/2 frees it at termination */
      return 1;
   } else { printf("DosSetMem: complete\n"); }

   strcpy(MyObject, "The memory object has just been used.");

              /* Check COMMIT status of the memory object. */

   ulMemSize = ulObjSize;
   rc = DosQueryMem(MyObject, &ulMemSize, &ulMemFlags);
   if (rc == NO_ERROR) {
      if (ulMemFlags & PAG_COMMIT) {
         printf("   Page containing MyObject is now committed.\n");
      } else {
         printf("Error: Page containing MyObject has not
                 been committed.\n");
      } /* endif */
   } else {
      printf("DosQueryMem error: return code = %u\n",rc);
   } /* endif */

   rc = DosFreeMem(MyObject);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error: return code = %u\n",rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosAllocMem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosAllocSharedMem

DosAllocSharedMem - Syntax

Allocates a shared memory object within the virtual-address space.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PPVOID    ppb;      /*  A pointer to a variable that will receive the base address of the allocated range of pag
PSZ       pszName;  /*  An optional address of the name string associated with the shared memory object to be al
ULONG     cb;       /*  Size, in bytes, of the shared memory object to allocate. */
ULONG     flag;     /*  Allocation attribute and desired access protection flags. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosAllocSharedMem(ppb, pszName, cb,
         flag);
```

DosAllocSharedMem Parameter - ppb

ppb (PPVOID) - output
A pointer to a variable that will receive the base address of the allocated range of pages.

OS/2 determines where to allocate the virtual address for the shared memory object.

DosAllocSharedMem Parameter - pszName

pszName (PSZ) - input
An optional address of the name string associated with the shared memory object to be allocated.

The name is an ASCIIZ string in the format of an OS/2 file name, and is in the subdirectory, \SHAREMEM\; for example, \SHAREMEM\PUBLIC.DAT.

To allocate unnamed shared memory, set this parameter to NULL. If you want to use unnamed shared memory, the *flag* parameter must include either OBJ_GETTABLE or OBJ_GIVEABLE.

DosAllocSharedMem Parameter - cb

cb (ULONG) - input
Size, in bytes, of the shared memory object to allocate.

The size is rounded up to the next page-size boundary. The size of a page is 4KB.

DosAllocSharedMem Parameter - flag

flag (ULONG) - input
Allocation attribute and desired access protection flags.

A set of flags describing the allocation attributes and desired access protection for the shared memory object. Possible values are shown in the following lists:

Allocation Attributes

PAG_COMMIT (0x00000010)
All pages in the shared memory object are initially committed.

OBJ_GIVEABLE (0x00000200)
The access to the memory object can be given to another process using the DosGiveSharedMem function.

OBJ_GETTABLE (0x00000100)
The memory object can be accessed by another process that knows the address of the memory and calls the DosGetSharedMem function.

OBJ_TILE (0x00000040)
The shared memory object must be allocated in the first 512MB of virtual-address space, with 16-bit selectors mapping the memory object.

The 16-bit selectors are allocated to map the 32-bit object at 64KB boundaries. The figure in the description of the **Parameters** for DosAllocMem shows how the 16-bit alias selectors map the 32-bit object. **Desired Access Protection**

PAG_EXECUTE (0x00000004)
Execute access to the committed pages in the private memory object is desired.

PAG_READ (0x00000001)
Read access is desired.

PAG_WRITE (0x00000002)
Write access is desired.

PAG_GUARD (0x00000008),
Access to the committed pages in the private memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits of PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set.

DosAllocSharedMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosAllocSharedMem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
123	ERROR_INVALID_NAME
183	ERROR_ALREADY_EXISTS

For a full list of error codes, see Errors.

DosAllocSharedMem - Parameters

ppb (PPVOID) - output
A pointer to a variable that will receive the base address of the allocated range of pages.

OS/2 determines where to allocate the virtual address for the shared memory object.

pszName (PSZ) - input
An optional address of the name string associated with the shared memory object to be allocated.

The name is an ASCIIZ string in the format of an OS/2 file name, and is in the subdirectory, \SHAREMEM\; for example, \SHAREMEM\PUBLIC.DAT.

To allocate unnamed shared memory, set this parameter to NULL. If you want to use unnamed shared memory, the *flag* parameter must include either OBJ_GETTABLE or OBJ_GIVEABLE.

cb (ULONG) - input
Size, in bytes, of the shared memory object to allocate.

The size is rounded up to the next page-size boundary. The size of a page is 4KB.

flag (ULONG) - input
Allocation attribute and desired access protection flags.

A set of flags describing the allocation attributes and desired access protection for the shared memory object. Possible values are shown in the following lists:

Allocation Attributes

PAG_COMMIT (0x00000010)
All pages in the shared memory object are initially committed.

OBJ_GIVEABLE (0x00000200)
The access to the memory object can be given to another process using the DosGiveSharedMem function.

OBJ_GETTABLE (0x00000100)
The memory object can be accessed by another process that knows the address of the memory and calls the DosGetSharedMem function.

OBJ_TILE (0x00000040)
The shared memory object must be allocated in the first 512MB of virtual-address space, with 16-bit selectors mapping the memory object.

The 16-bit selectors are allocated to map the 32-bit object at 64KB boundaries. The figure in the description of the **Parameters** for DosAllocMem shows how the 16-bit alias selectors map the 32-bit object. **Desired Access Protection**

PAG_EXECUTE (0x00000004)
Execute access to the committed pages in the private memory object is desired.

PAG_READ (0x00000001)
Read access is desired.

PAG_WRITE (0x00000002)
Write access is desired.

PAG_GUARD (0x00000008),
Access to the committed pages in the private memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits of PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set.

ulrc (APIRET) - returns
Return Code.

DosAllocSharedMem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
123	ERROR_INVALID_NAME
183	ERROR_ALREADY_EXISTS

For a full list of error codes, see Errors.

DosAllocSharedMem - Remarks

DosAllocSharedMem allocates a shared memory object within the virtual-address space.

Allocating a shared memory object causes the creation of an object that describes a region of memory that can be shared. The virtual-address space in the calling process is allocated and mapped to the shared memory object.

The virtual-address space for a shared memory object is reserved at the same location in the virtual address space of every process. This allows any process to gain access to the shared object at the same virtual address where it was originally allocated.

When the shared memory object is given a name, the shared memory object can be shared by other processes that gain access through the shared memory name (see DosGetNamedSharedMem).

To specify the name for the shared memory object, the name string provided must include the prefix "\SHAREMEM\".

It is an error to request giveable or gettable named shared memory.

To allocate unnamed shared memory, set the *pszName* parameter to NULL.

If the shared memory object is unnamed, it may be specified as giveable (OBJ_GIVEABLE) or gettable (OBJ_GETTABLE). Unnamed shared memory may be shared by all processes that get access to the shared memory object (see DosGetSharedMem), or are given access to the shared memory object (see DosGiveSharedMem).

It is an error to request non-giveable or non-gettable unnamed shared memory.

The allocated memory object is rounded up to a multiple of 4KB in size.

The committed memory allocated by DosAllocSharedMem is movable and can be swapped.

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

The tiled allocation attribute is provided for compatibility with the existing 16-bit implementation of the operating system. If the shared memory object is tiled, the virtual address for the shared memory object will be within the first 512MB of the virtual address space, with 16-bit selectors mapping the memory object.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded up to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocSharedMem call with a size of 1, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosAllocSharedMem - Related Functions

Related Functions

- DosAllocMem
- DosFreeMem
- DosGetNamedSharedMem
- DosGetSharedMem
- DosGiveSharedMem

DosAllocSharedMem - Example Code

This example allocates named shared memory as Read/Write, and commits it during allocation. It also writes to it, and show how other processes can access it.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main (VOID)
{
   PVOID  pvShrObject = NULL;      /* Pointer to shared memory object      */
   PSZ    pszMemName  = "\\SHAREMEM\\MYTOOL\\APPLICAT.DAT"; /* Object name */
   PVOID  pvAltObject = NULL;      /* Alternate pointer to shared memory   */
   APIRET rc          = NO_ERROR;  /* Return code                          */
   ULONG  ulObjSize   = 1024;   /* Size (system rounds to 4096 - page bdy) */

   rc = DosAllocSharedMem(&pvShrObject,   /* Pointer to object pointer     */
                          pszMemName,     /* Name for shared memory        */
                          ulObjSize,      /* Desired size of object        */
                          PAG_COMMIT |    /* Commit memory                 */
                          PAG_WRITE );    /* Allocate memory as read/write */
   if (rc != NO_ERROR) {
      printf("DosAllocSharedMem error:  return code = %u\n",rc);
      return 1;
   }

   strcpy(pvShrObject, "Write your shared application data here.");

      /* Get the address of the shared memory and reference it that way.
         (Done for illustrative purposes only, this is how another process
         would go about accessing the named shared memory.)            */

   rc = DosGetNamedSharedMem(&pvAltObject,  /* Pointer to pointer of object */
                             pszMemName,    /* Name of shared memory        */
                             PAG_READ);     /* Want read-only access        */
   if (rc != NO_ERROR) {
      printf("DosGetNamedSharedMem error:  return code = %u\n",rc);
      return 1;
   }

   printf("Shared data read was \"%s\"\n",pvAltObject);

   rc = DosFreeMem(pvShrObject);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error:  return code = %u\n",rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosAllocSharedMem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosAllocThreadLocalMemory

DosAllocThreadLocalMemory - Syntax

Allocates a block of memory that is unique, or local, to a thread.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG       cb;  /*  Number of 4-byte DWORDs to allocate. */
PULONG     *p;   /*  Address of the memory block returned by the function. */
APIRET      rc;  /*  Return Code. */

rc = DosAllocThreadLocalMemory(cb, p);
```

DosAllocThreadLocalMemory Parameter - cb

cb (ULONG) - input
Number of 4-byte DWORDs to allocate.

Up to 8 DWORDs (32 bytes) can be requested each time this function is called. A maximum of 32 DWORDs (128 bytes) can be allocated in total.

DosAllocThreadLocalMemory Parameter - p

p (PULONG *) - output
Address of the memory block returned by the function.

DosAllocThreadLocalMemory Return Value - rc

rc (APIRET) - returns
Return Code.

This function returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosAllocThreadLocalMemory - Parameters

cb (ULONG) - input
Number of 4-byte DWORDs to allocate.

Up to 8 DWORDs (32 bytes) can be requested each time this function is called. A maximum of 32 DWORDs (128 bytes) can be allocated in total.

p (PULONG *) - output
Address of the memory block returned by the function.

rc (APIRET) - returns
Return Code.

This function returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosAllocThreadLocalMemory - Remarks

When a process is started, a small block of memory is set aside to be used as a thread-local memory area. This memory is at the same virtual address for each thread, but is backed by different physical memory. This permits each thread to have a small block of memory that is unique, or local, to that thread.

The thread-local memory area consists of 32 DWORDs (128 bytes), each DWORD being 4 bytes in size. Up to 8 DWORDs (32 bytes) can be requested each time this function is called. If you want to allocate more than 8 DWORDs, you must call this function more than once.

Allocation is by DWORD only. If you want to store a BYTE in the thread-local memory area, you would still allocate a DWORD, then store the BYTE in it.

DosAllocThreadLocalMemory - Related Functions

Related Functions

- DosFreeThreadLocalMemory

DosAllocThreadLocalMemory - Example Code

This example shows how to allocate and free 3 WORDs of local thread memory.

```
#define  INCL_DOSPROCESS
#define  INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>          /* Needed for printf */

PULONG   pulThreadDWords = NULL;     /* Pointer to thread DWORDs returned */
APIRET   rc              = NO_ERROR; /* Return code */

int main(VOID) {

         /* Allocate 3 DWORDs of local thread memory */

  rc = DosAllocThreadLocalMemory(3,                   /* Number of DWORDs */
                                 &pulThreadDWords);   /* Address returned */

  if (rc != NO_ERROR) {
    printf("DosAllocThreadLocalMemory error: return code = %u\n", rc);
    return 1;
  }

          /* ... Use the thread-local memory ... */

  rc = DosFreeThreadLocalMemory(pulThreadDWords);     /* Free the DWORDs */

  if (rc != NO_ERROR) {
    printf("DosFreeThreadLocalMemory error: return code = %u\n", rc);
    return 1;
  }

  return NO_ERROR;
}
```

DosAllocThreadLocalMemory - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosAsyncTimer

DosAsyncTimer - Syntax

Starts an asynchronous, single-interval timer.

```
#define INCL_DOSDATETIME
#include <os2.h>

ULONG      msec;     /*  The time, in milliseconds, before the event semaphore specified by hsem is posted. */
HSEM       hsem;     /*  The handle of an event semaphore that will be posted when the time specified by msec ha
PHTIMER    phtimer;  /*  A pointer to the timer handle. */
APIRET     ulrc;     /*  Return Code. */

ulrc = DosAsyncTimer(msec, hsem, phtimer);
```

DosAsyncTimer Parameter - msec

msec (ULONG) - input
The time, in milliseconds, before the event semaphore specified by *hsem* is posted.

(The system rounds this value up to the next clock tick.)

DosAsyncTimer Parameter - hsem

hsem (HSEM) - input
The handle of an event semaphore that will be posted when the time specified by *msec* has elapsed.

This semaphore must be a shared event semaphore, and should be reset before issuing DosAsyncTimer.

DosAsyncTimer Parameter - phtimer

phtimer (PHTIMER) - output
A pointer to the timer handle.

This handle can be passed to DosStopTimer to stop the timer before its time interval expires.

DosAsyncTimer Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosAsyncTimer returns one of the following values:

0	NO_ERROR
323	ERROR_TS_SEMHANDLE
324	ERROR_TS_NOTIMER

For a full list of error codes, see Errors.

DosAsyncTimer - Parameters

msec (ULONG) - input
The time, in milliseconds, before the event semaphore specified by *hsem* is posted.

(The system rounds this value up to the next clock tick.)

hsem (HSEM) - input
The handle of an event semaphore that will be posted when the time specified by *msec* has elapsed.

This semaphore must be a shared event semaphore, and should be reset before issuing DosAsyncTimer.

phtimer (PHTIMER) - output
A pointer to the timer handle.

This handle can be passed to DosStopTimer to stop the timer before its time interval expires.

ulrc (APIRET) - returns
Return Code.

DosAsyncTimer returns one of the following values:

0	NO_ERROR
323	ERROR_TS_SEMHANDLE
324	ERROR_TS_NOTIMER

For a full list of error codes, see Errors.

DosAsyncTimer - Remarks

DosAsyncTimer starts a single-interval timer. The timer runs asynchronously to the calling thread, and posts an event semaphore when the specified time interval expires.

Time intervals for DosAsyncTimer, DosStartTimer, and DosSleep are specified in milliseconds; however, it is important to recognize that the actual duration of the specified time interval will be affected by two factors:

- First, the system clock keeps track of time in less precise units known as *clock ticks*. The duration of a clock tick depends on the frequency of the system-clock interrupt that is used by your computer. (To determine the duration of the clock tick on your computer, issue DosQuerySysInfo and examine the timer-interval field.)

 Because clock ticks are less precise than millisecond values, any time interval that is specified in milliseconds will be rounded up to the next clock tick.

- Second, because the system is a priority-based, multitasking operating system, there is no guarantee that a thread will resume immediately after the timer interval expires. If a higher-priority process or thread is running, the timed thread blocks. (To minimize the inaccuracy caused by preemptive scheduling, an application can dedicate a thread to managing time-critical tasks, and then raise that thread to a higher priority.)

These factors usually cause the timer interval to be longer than requested; however, it will generally be within a few clock ticks.

DosAsyncTimer - Related Functions

Related Functions

- DosCreateEventSem
- DosGetDateTime
- DosOpenEventSem

- DosResetEventSem
- DosSetDateTime
- DosSleep
- DosStartTimer
- DosStopTimer
- DosWaitEventSem

DosAsyncTimer - Example Code

This example sets the asyncronous timer to 7 seconds. After this time, it posts a message to an event semaphore.

```
 #define INCL_DOSSEMAPHORES   /* Semaphore values */
 #define INCL_DOSDATETIME     /* Timer support    */
 #define INCL_DOSERRORS       /* DOS error values */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 PSZ     szSemName  = "\\SEM32\\TIMER\\THREAD1\\EVENT1"; /* Semaphore name */
 HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
 HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
 APIRET  rc            = NO_ERROR;           /* Return code               */

    rc = DosCreateEventSem(szSemName,      /* Name of semaphore to create  */
                           &hevEvent1,     /* Handle of semaphore returned */
                           DC_SEM_SHARED,  /* Shared semaphore             */
                           FALSE);         /* Semaphore is in RESET state  */
    if (rc != NO_ERROR) {
        printf("DosCreateEventSem error: return code = %u\n", rc);
        return 1;       }

    rc = DosAsyncTimer(7000L,              /* 7 second interval            */
                       (HSEM) hevEvent1,   /* Semaphore to post            */
                       &htimerEvent1);     /* Timer handler (returned)     */
    if (rc != NO_ERROR) {
        printf("DosAsyncTimer error: return code = %u\n", rc);
        return 1;
    } else {
        printf("Timer will expire in about 7 seconds...\n");
    }

           /* ... add your other processing here... */

    rc = DosWaitEventSem(hevEvent1,            /* Wait for AsyncTimer event */
                 (ULONG) SEM_INDEFINITE_WAIT); /* As long as it takes       */
    if (rc != NO_ERROR) {
        printf("DosWaitEventSem error: return code = %u\n", rc);
        return 1;
    }

    rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u", rc);
        return 1;
    }

return NO_ERROR;
}
```

DosAsyncTimer - Topics

Select an item:
Syntax
Parameters
Returns

Remarks
Example Code
Related Functions
Glossary

DosBeep

DosBeep - Syntax

Generates sound from the speaker.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     freq;  /*  Cycles per second (Hertz) in the range of 0x25 to 0x7FFF. */
ULONG     dur;   /*  The length of the sound in milliseconds. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosBeep(freq, dur);
```

DosBeep Parameter - freq

freq (ULONG) - input
Cycles per second (Hertz) in the range of 0x25 to 0x7FFF.

DosBeep Parameter - dur

dur (ULONG) - input
The length of the sound in milliseconds.

DosBeep Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosBeep returns one of the following values:

0	NO_ERROR
395	ERROR_INVALID_FREQUENCY

For a full list of error codes, see Errors.

DosBeep - Parameters

freq (ULONG) - input
Cycles per second (Hertz) in the range of 0x25 to 0x7FFF.

dur (ULONG) - input
The length of the sound in milliseconds.

ulrc (APIRET) - returns
Return Code.

DosBeep returns one of the following values:

0	NO_ERROR
395	ERROR_INVALID_FREQUENCY

For a full list of error codes, see Errors.

DosBeep - Related Functions

Related Functions

- DosDevConfig
- DosDevIOCtl
- DosPhysicalDisk

DosBeep - Example Code

This example generates a beep for one second at a frequency of 1440 Hz.

```
#define INCL_DOSPROCESS    /* DOS Process and thread values */
#define INCL_DOSERRORS     /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

   APIRET   rc = NO_ERROR;     /* Return code                        */

   rc = DosBeep(1440L,         /* Beep frequency, in hertz           */
                1000L);        /* Duration of beep, in milliseconds */

   if (rc != NO_ERROR) {
      printf("DosBeep error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosBeep - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosCallNPipe

DosCallNPipe - Syntax

Makes a procedure call to a duplex message pipe.

```
#define INCL_DOSNMPIPES
#include <os2.h>

PSZ       pszName;    /*  The ASCIIZ name of the pipe to be opened. */
PVOID     pInbuf;     /*  A pointer to the buffer that is to be written to the pipe. */
ULONG     cbIn;       /*  The number of bytes to be written. */
PVOID     pOutbuf;    /*  A pointer to the buffer for returned data. */
ULONG     cbOut;      /*  The maximum size, in bytes, of returned data. */
PULONG    pcbActual;  /*  A pointer to the ULONG in which the number of bytes actually read is returned. */
ULONG     msec;       /*  The maximum time, in milliseconds, to wait for a pipe instance to become available. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosCallNPipe(pszName, pInbuf, cbIn,
         pOutbuf, cbOut, pcbActual, msec);
```

DosCallNPipe Parameter - pszName

pszName (PSZ) - input
The ASCIIZ name of the pipe to be opened.

Pipe names must include the prefix \PIPE\ and must conform to file-system naming conventions. When communicating with a remote process, the computer name must also be included, using the format \\ComputerName\PIPE\FileName.

DosCallNPipe Parameter - pInbuf

pInbuf (PVOID) - input
A pointer to the buffer that is to be written to the pipe.

DosCallNPipe Parameter - cbIn

cbIn (ULONG) - input
The number of bytes to be written.

DosCallNPipe Parameter - pOutbuf

pOutbuf (PVOID) - output
A pointer to the buffer for returned data.

DosCallNPipe Parameter - cbOut

cbOut (ULONG) - input
The maximum size, in bytes, of returned data.

DosCallNPipe Parameter - pcbActual

pcbActual (PULONG) - output
A pointer to the ULONG in which the number of bytes actually read is returned.

DosCallNPipe Parameter - msec

msec (ULONG) - input
The maximum time, in milliseconds, to wait for a pipe instance to become available.

DosCallNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCallNPipe returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED

11	ERROR_BAD_FORMAT
95	ERROR_INTERRUPT
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED
234	ERROR_MORE_DATA

On the PowerPC platform, DosCallNPipe fails with error code ERROR_PATH_NOT_FOUND when the pipe name contains more than one \ in the name. For example, the following pipe name would cause DosCallNPipe to fail:

```
\pipe\\\\\longname.nam
```

For a full list of error codes, see Errors.

DosCallNPipe - Parameters

pszName (PSZ) - input
The ASCIIZ name of the pipe to be opened.

Pipe names must include the prefix \PIPE\ and must conform to file-system naming conventions. When communicating with a remote process, the computer name must also be included, using the format \\ComputerName\PIPE\FileName.

pInbuf (PVOID) - input
A pointer to the buffer that is to be written to the pipe.

cbIn (ULONG) - input
The number of bytes to be written.

pOutbuf (PVOID) - output
A pointer to the buffer for returned data.

cbOut (ULONG) - input
The maximum size, in bytes, of returned data.

pcbActual (PULONG) - output
A pointer to the ULONG in which the number of bytes actually read is returned.

msec (ULONG) - input
The maximum time, in milliseconds, to wait for a pipe instance to become available.

ulrc (APIRET) - returns
Return Code.

DosCallNPipe returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
11	ERROR_BAD_FORMAT
95	ERROR_INTERRUPT
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED
234	ERROR_MORE_DATA

On the PowerPC platform, DosCallNPipe fails with error code ERROR_PATH_NOT_FOUND when the pipe name contains more than one \ in the name. For example, the following pipe name would cause DosCallNPipe to fail:

```
\pipe\\\\\longname.nam
```

For a full list of error codes, see Errors.

DosCallNPipe - Remarks

DosCallNPipe combines the functions of DosOpen, DosTransactNPipe, and DosClose for a duplex message pipe. If no instances of a pipe are available, DosCallNPipe waits for a specified time interval, and returns ERROR_INTERRUPT if the time interval elapses.

If this function is issued for a pipe that is not a duplex message pipe, ERROR_BAD_FORMAT is returned.

If an invalid pipe name is specified, DosCallNPipe returns ERROR_FILE_NOT_FOUND.

If *pOutbuf* is too small to contain the response message, ERROR_MORE_DATA is returned.

If the server process has not issued DosConnectNPipe to put the pipe into a listening state, DosCallNPipe returns ERROR_PIPE_BUSY.

Clients of named pipes created with the NP_ACCESS_OUTBOUND or NP_ACCESS_INBOUND access mode cannot use the DosCallNPipe function. If the named pipe's client uses the DosCallNPipe function, the function returns error code ERROR_ACCESS_DENIED.

ERROR_BAD_PIPE is returned if you specify an invalid name or file handle.

DosCallNPipe - Related Functions

Related Functions

- DosConnectNPipe
- DosClose
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosCallNPipe - Example Code

This example shows how to make a call to an existing named pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     outmsg[256]           = "";         /* Output message buffer */
   CHAR     inmsg[256]            = "";         /* Input message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   PIPEINFO PipeBuffer[4]         = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
```

```
   PIPESEMSTATE infobuf[3]         = {{0}};

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);   /* Make above printf show on display */
   gets(outmsg);

   rc = DosCallNPipe("\\PIPE\\EXAMPLE",   /* Name of duplex pipe */
                     outmsg,              /* Output message buffer */
                     strlen(outmsg),      /* Size of output message */
                     inmsg,               /* Input message buffer */
                     sizeof(inmsg),       /* Size of input buffer */
                     &bytes,              /* Number of bytes read */
                     30000L);             /* Wait 30 seconds for pipe */
   if (rc != NO_ERROR) {
      printf("DosCallNPipe error: error code = %u\n", rc);
      return 1;
   } else {
      printf("\nMessage received from PIPEHOST: %s\n\n", inmsg);
   } /* endif */

   return NO_ERROR;
}
```

DosCallNPipe - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCancelLockRequest

DosCancelLockRequest - Syntax

Cancels an outstanding DosSetFileLocks request.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE        hFile;    /*  File handle used in the DosSetFileLocks function that is to be cancelled. */
PFILELOCK    pflLock;  /*  Address of the structure describing the region to be locked by DosSetFileLocks. */
APIRET       ulrc;     /*  Return Code. */

ulrc = DosCancelLockRequest(hFile, pflLock);
```

DosCancelLockRequest Parameter - hFile

hFile (HFILE) - input
File handle used in the DosSetFileLocks function that is to be cancelled.

DosCancelLockRequest Parameter - pflLock

pflLock (PFILELOCK) - input
Address of the structure describing the region to be locked by DosSetFileLocks.

DosCancelLockRequest Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCancelLockRequest returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
173	ERROR_CANCEL_VIOLATION

For a full list of error codes, see Errors.

DosCancelLockRequest - Parameters

hFile (HFILE) - input
File handle used in the DosSetFileLocks function that is to be cancelled.

pflLock (PFILELOCK) - input
Address of the structure describing the region to be locked by DosSetFileLocks.

ulrc (APIRET) - returns
Return Code.

DosCancelLockRequest returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
173	ERROR_CANCEL_VIOLATION

For a full list of error codes, see Errors.

DosCancelLockRequest - Remarks

DosCancelLockRequest allows a process to cancel the lock range request of an outstanding DosSetFileLocks function.

If two threads in a process are waiting on a lock file range, and another thread issues DosCancelLockRequest for that lock file range, then both waiting threads are released.

Not all file-system drivers (FSDs) can cancel an outstanding lock request.

Local Area Network (LAN) servers cannot cancel an outstanding lock request if they use a version of the operating system prior to OS/2 Version 2.00.

DosCancelLockRequest - Related Functions

Related Functions

- DosSetFileLocks

DosCancelLockRequest - Example Code

This example opens a file named "CANLOCK.DAT", locks a block of the data, writes some data to it, and then cancels the lock request.

```
#define INCL_DOSFILEMGR       /* File Manager values */
#define INCL_DOSERRORS        /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

HFILE     FileHandle   = NULLHANDLE;  /* File handle */
ULONG     Action       = 0,           /* Action taken by DosOpen */
          Wrote        = 0;           /* Number of bytes written by DosWrite */
CHAR      FileData[40] = "Forty bytes of demonstration text data\r\n";
APIRET    rc           = NO_ERROR;    /* Return code */

FILELOCK  LockArea     = {0},         /* Area of file to lock */
          UnlockArea   = {0};         /* Area of file to unlock */

rc = DosOpen("canlock.dat",                 /* File to open */
             &FileHandle,                   /* File handle */
             &Action,                       /* Action taken */
             256L,                          /* File primary allocation */
             FILE_ARCHIVED,                 /* File attributes */
             FILE_OPEN | FILE_CREATE,       /* Open function type */
             OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
             0L);                           /* No extended attributes */
if (rc != NO_ERROR) {                       /* If open failed */
   printf("DosOpen error: return code = %u\n", rc);
   return 1; }

LockArea.lOffset = 0L;              /* Start locking at beginning of file */
LockArea.lRange =  40L;             /* Use a lock range of 40 bytes       */

rc = DosSetFileLocks(FileHandle,        /* File handle   */
                     &UnlockArea,       /* No unlock area */
                     &LockArea,         /* Lock current record */
                     2000L,             /* Lock time-out value of 2 seconds */
                     0L);               /* Exclusive lock, not atomic */
if (rc != NO_ERROR) {
   printf("DosSetFileLocks error: return code = %u\n", rc);
   return 1;
}

rc = DosWrite(FileHandle, FileData, sizeof(FileData), &Wrote);
if (rc != NO_ERROR) {
   printf("DosWrite error: return code = %u\n", rc);
   return 1;
}
/* Should check if (rc != NO_ERROR) here... */

rc = DosCancelLockRequest(FileHandle, &LockArea);
```

```
if (rc != NO_ERROR) {
   printf("DosCancelLockRequest error: return code = %u\n", rc);
   return 1;
}
rc = DosClose(FileHandle);
/* Should check if (rc != NO_ERROR) here... */

return NO_ERROR;
}
```

DosCancelLockRequest - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosClose

DosClose - Syntax

Closes a handle to a file, pipe, or device.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;  /*  The handle returned by a previous call to DosCreateNPipe, DosCreatePipe, DosDupHandle, or
APIRET    ulrc;   /*  Return Code. */

ulrc = DosClose(hFile);
```

DosClose Parameter - hFile

hFile (HFILE) - input
The handle returned by a previous call to DosCreateNPipe, DosCreatePipe, DosDupHandle, or DosOpen.

DosClose Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosClose returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosClose - Parameters

hFile (HFILE) - input
The handle returned by a previous call to DosCreateNPipe, DosCreatePipe, DosDupHandle, or DosOpen.

ulrc (APIRET) - returns
Return Code.

DosClose returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosClose - Remarks

Issuing DosClose with the handle to a file closes a handle to a file, pipe, or device.

If additional handles to a file were created with DosDupHandle, DosClose must be issued for the duplicate handles before the directory is updated, and information in internal buffers is written to the medium.

Closing a device handle causes the device to be notified of the close, if appropriate.

Named-Pipe Considerations

DosClose closes a named pipe by handle. When all handles that refer to one end of a pipe are closed, the pipe is considered broken.

If the client end closes, no other process can reopen the pipe until the serving end issues DosDisConnectNPipe, followed by DosConnectNPipe.

If the server end closes when the pipe is already broken, the pipe is immediately deallocated; otherwise, it is not deallocated until the last client handle is closed.

DosClose - Related Functions

Related Functions

- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle

- DosOpen
- DosResetBuffer

DosClose - Example Code

This example opens, or creates and opens, a file named "DOSTEST.DAT", writes to it, reads from it, and finally closes it.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(void) {
   HFILE  hfFileHandle   = 0L;  /* Handle for file being manipulated */
   ULONG  ulAction       = 0;   /* Action taken by DosOpen */
   ULONG  ulBytesRead    = 0;   /* Number of bytes read by DosRead */
   ULONG  ulWrote        = 0;   /* Number of bytes written by DosWrite */
   ULONG  ulLocal        = 0;   /* File pointer position after DosSetFilePtr */
   UCHAR  uchFileName[20]  = "dostest.dat",  /* Name of file */
          uchFileData[100] = " ";            /* Data to write to file */
   APIRET rc             = NO_ERROR;         /* Return code */

   /* Open the file dostest.dat. Use an existing file or create a new */
   /* one if it doesn't exist.                                        */
   rc = DosOpen(uchFileName,                    /* File path name */
                &hfFileHandle,                  /* File handle */
                &ulAction,                      /* Action taken */
                100L,                           /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
                OPEN_ACTION_CREATE_IF_NEW |
                OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type */
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file */
                0L);                            /* No extended attribute */

   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosOpen: Action taken = %ld\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n1...\n2...\n3\n");
   rc = DosWrite (hfFileHandle,              /* File handle */
                  (PVOID) uchFileData,       /* String to be written */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote);                 /* Bytes actually written */

   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosSetFilePtr (hfFileHandle,           /* File Handle */
                       0L,                     /* Offset */
                       FILE_BEGIN,             /* Move from BOF */
                       &ulLocal);              /* New location address */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosRead (hfFileHandle,                /* File Handle */
                 uchFileData,                 /* String to be read */
                 100L,                        /* Length of string to be read */
                 &ulBytesRead);               /* Bytes actually read */

   if (rc != NO_ERROR) {
```

```
      printf("DosRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosClose(hfFileHandle);                /* Close the file */

   if (rc != NO_ERROR) {
      printf("DosClose error: return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosClose - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCloseEventSem

DosCloseEventSem - Syntax

Closes an event semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HEV       hev;   /*  The handle of the event semaphore to close. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosCloseEventSem(hev);
```

DosCloseEventSem Parameter - hev

hev (HEV) - input
The handle of the event semaphore to close.

DosCloseEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCloseEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
301	ERROR_SEM_BUSY

For a full list of error codes, see Errors.

DosCloseEventSem - Parameters

hev (HEV) - input
The handle of the event semaphore to close.

ulrc (APIRET) - returns
Return Code.

DosCloseEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
301	ERROR_SEM_BUSY

For a full list of error codes, see Errors.

DosCloseEventSem - Related Functions

Related Functions

- DosCreateEventSem
- DosOpenEventSem
- DosPostEventSem
- DosQueryEventSem
- DosResetEventSem
- DosWaitEventSem

DosCloseEventSem - Example Code

This example creates an event semaphore and the asyncronous timer uses it to post when its time expires. Finally, it closes the event semaphore.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
```

```
 PSZ     szSemName  = "\\SEM32\\TIMER\\THREAD1\\EVENT1"; /* Semaphore name */
 HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
 HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
 APIRET  rc            = NO_ERROR;           /* Return code               */

    rc = DosCreateEventSem(szSemName,      /* Name of semaphore to create  */
                           &hevEvent1,     /* Handle of semaphore returned */
                           DC_SEM_SHARED,  /* Shared semaphore             */
                           FALSE);         /* Semaphore is in RESET state  */
    if (rc != NO_ERROR) {
        printf("DosCreateEventSem error: return code = %u\n", rc);
        return 1;       }

    rc = DosAsyncTimer(7000L,              /* 7 second interval            */
                       (HSEM) hevEvent1,   /* Semaphore to post            */
                       &htimerEvent1);     /* Timer handler (returned)     */
    if (rc != NO_ERROR) {
        printf("DosAsyncTimer error: return code = %u\n", rc);
        return 1;
    } else {
        printf("Timer will expire in about 7 seconds...\n");
    }

           /* ... add your other processing here... */

    rc = DosWaitEventSem(hevEvent1,            /* Wait for AsyncTimer event */
                 (ULONG) SEM_INDEFINITE_WAIT); /* As long as it takes       */
    if (rc != NO_ERROR) {
        printf("DosWaitEventSem error: return code = %u\n", rc);
        return 1;
    }

    rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u", rc);
        return 1;
    }

return NO_ERROR;
}
```

DosCloseEventSem - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosCloseMutexSem

DosCloseMutexSem - Syntax

Closes a mutex semaphore.

```
#define INCL_DOSSEMAPHORES
```

```
#include <os2.h>

HMTX      hmtx;  /*  The handle of the mutex semaphore to close. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosCloseMutexSem(hmtx);
```

DosCloseMutexSem Parameter - hmtx

hmtx (HMTX) - input
The handle of the mutex semaphore to close.

DosCloseMutexSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCloseMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
301	ERROR_SEM_BUSY

For a full list of error codes, see Errors.

DosCloseMutexSem - Parameters

hmtx (HMTX) - input
The handle of the mutex semaphore to close.

ulrc (APIRET) - returns
Return Code.

DosCloseMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
301	ERROR_SEM_BUSY

For a full list of error codes, see Errors.

DosCloseMutexSem - Remarks

DosCloseMutexSem closes (ends access to) a mutex semaphore for all of the threads in the calling process.

When all of the processes that had opened the semaphore have either closed the semaphore or have ended, the semaphore is deleted by the system.

DosCloseMutexSem - Related Functions

Related Functions

- DosCreateMutexSem
- DosOpenMutexSem
- DosQueryMutexSem
- DosReleaseMutexSem
- DosRequestMutexSem

DosCloseMutexSem - Example Code

This example creates a mutual exclusion (Mutex) semaphore, manipulates it, and finally closes it.

```
#define INCL_DOSSEMAPHORES      /* Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HMTX    hmtx     = NULLHANDLE; /* Mutex semaphore handle */
PID     pidOwner = 0;          /* PID of current mutex semaphore owner */
TID     tidOwner = 0;          /* TID of current mutex semaphore owner */
ULONG   ulCount  = 0;          /* Request count for the semaphore */
APIRET  rc       = NO_ERROR;   /* Return code */

   rc = DosCreateMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                          &hmtx, 0, FALSE);       /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }
        /* This would normally be done by another unit of work */
   rc = DosOpenMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                        &hmtx);                 /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosRequestMutexSem(hmtx,                      /* Handle of semaphore */
                           (ULONG) SEM_INDEFINITE_WAIT);  /* Timeout (none) */
   if (rc != NO_ERROR) {
      printf("DosRequestMutexSem error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryMutexSem(hmtx,         /* Handle of semaphore */
                         &pidOwner,    /* Process ID of owner */
                         &tidOwner,    /* Thread ID of owner */
                         &ulCount);    /* Count */
   if (rc != NO_ERROR) {
      printf("DosQueryMutexSem error: return code = %u\n", rc);
      return 1;
   } else  {
      printf("Semaphore owned by PID %u, TID %u.", pidOwner, tidOwner);
      printf("  Request count is %u.\n", ulCount);
   } /* endif */

   rc = DosReleaseMutexSem(hmtx);        /* Relinquish ownership */
   if (rc != NO_ERROR) {
      printf("DosReleaseMutexSem error: return code = %u\n", rc);
      return 1;
    }
   rc = DosCloseMutexSem(hmtx);          /* Close mutex semaphore */
   if (rc != NO_ERROR) {
      printf("DosCloseMutexSem error: return code = %u\n", rc);
      return 1;
   }
```

```
return NO_ERROR;
}
```

DosCloseMutexSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCloseMuxWaitSem

DosCloseMuxWaitSem - Syntax

Closes a muxwait semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMUX      hmux;  /*  The handle of the muxwait semaphore to close. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosCloseMuxWaitSem(hmux);
```

DosCloseMuxWaitSem Parameter - hmux

hmux (HMUX) - input
The handle of the muxwait semaphore to close.

DosCloseMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCloseMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
301	ERROR_SEM_BUSY

For a full list of error codes, see Errors.

DosCloseMuxWaitSem - Parameters

hmux (HMUX) - input
The handle of the muxwait semaphore to close.

ulrc (APIRET) - returns
Return Code.

DosCloseMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
301	ERROR_SEM_BUSY

For a full list of error codes, see Errors.

DosCloseMuxWaitSem - Remarks

DosCloseMuxWaitSem closes (ends access to) a muxwait semaphore for all of the threads in the calling process.

When all of the processes that opened the semaphore have either closed the semaphore or have ended, the system deletes the semaphore.

DosCloseMuxWaitSem - Related Functions

Related Functions

- DosAddMuxWaitSem
- DosCreateMuxWaitSem
- DosDeleteMuxWaitSem
- DosOpenMuxWaitSem
- DosQueryMuxWaitSem
- DosWaitMuxWaitSem

DosCloseMuxWaitSem - Example Code

This example creates two semaphore record lists, with two event semaphores in each. It then creates a muxwait semaphore with the first semaphore record list, and it adds the second list later using DosAddMuxWaitSem.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX      hmuxHandAny = NULLHANDLE;      /* Muxwaithandle */
```

```
 HEV       hev[5]      = {0};             /* Event semaphores */
 SEMRECORD apsr[5]     = {{0}};           /* Semaphore records */
 APIRET    rc          = NO_ERROR;        /* Return code */
 ULONG     ulLoop      = 0;               /* Loop count */
 ULONG     ulSem       = 0;

for (ulLoop = 0; ulLoop < 5; ulLoop++) {
    rc = DosCreateEventSem((PSZ) NULL,
                           &hev[ulLoop],
                           0,
                           FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
    apsr[ulLoop].hsemCur = (HSEM) hev[ulLoop],
    apsr[ulLoop].ulUser = 0;
} /* endfor */

rc = DosCreateMuxWaitSem((PSZ) NULL,
                         &hmuxHandAny,
                         5L,               /* Number of semaphores in list */
                         apsr,             /* Semaphore list */
                         DCMW_WAIT_ANY);   /* Wait for any semaphore */
    if (rc != NO_ERROR) {
      printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosWaitMuxWaitSem(hmuxHandAny,
                       SEM_IMMEDIATE_RETURN,
                       &ulSem);           /* No semaphores have been posted, so
                                             we should see a timeout below...    */
    if (rc != ERROR_TIMEOUT) {
      printf("DosWaitMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosDeleteMuxWaitSem(hmuxHandAny, apsr[0].hsemCur);
    if (rc != NO_ERROR) {
      printf("DosDeleteMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }
rc = DosCloseMuxWaitSem(hmuxHandAny);
     if (rc != NO_ERROR) {
       printf("DosCloseMuxWaitSem error: return code = %u\n", rc);
       return 1;
     }

return NO_ERROR;
}
```

DosCloseMuxWaitSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCloseQueue

DosCloseQueue - Syntax

Ends access to a queue, or deletes a queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

HQUEUE    hq;    /*  The handle of the queue to be closed. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosCloseQueue(hq);
```

DosCloseQueue Parameter - hq

hq (HQUEUE) - input
The handle of the queue to be closed.

This is the handle received from a previous call to DosCreateQueue or DosOpenQueue.

DosCloseQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCloseQueue returns one of the following values:

0	NO_ERROR
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosCloseQueue - Parameters

hq (HQUEUE) - input
The handle of the queue to be closed.

This is the handle received from a previous call to DosCreateQueue or DosOpenQueue.

ulrc (APIRET) - returns
Return Code.

DosCloseQueue returns one of the following values:

0	NO_ERROR
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosCloseQueue - Remarks

DosCloseQueue ends further processing of a queue by the requesting process. The action taken depends on whether the requester is the owner of the queue or a writer of the queue.

For the owner, any outstanding elements are deleted. Other processes that have the queue open will receive the ERROR_QUE_INVALID_HANDLE return code on their next request.

For a writer of the queue, access to the queue is ended, but the queue elements are not deleted.

DosCloseQueue - Related Functions

Related Functions

- DosCreateQueue
- DosOpenQueue
- DosPeekQueue
- DosPurgeQueue
- DosQueryQueue
- DosReadQueue
- DosWriteQueue

DosCloseQueue - Example Code

This example creates and opens a queue named "SPECIAL.QUE", writes to it, reads from it, and finally closes it.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSPROCESS  /* DOS thread and process values */
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

   PSZ         szQueueName  = "\\QUEUES\\SPECIAL.QUE";
   HQUEUE      hqSpecialQue = NULLHANDLE; /* Queue handle                    */
   PSZ         DataBuffer   = "";         /* Data buffer for queue data      */
   REQUESTDATA Request      = {0};        /* Reques */
   PID         pidOwner     = 0;
   ULONG       ulDataLen    = 0;          /* Length of data returned         */
   BYTE        ElemPrty     = 0;          /* Priority of element (returned)  */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateQueue(&hqSpecialQue,    /* Queue handle                    */
             QUE_FIFO |                  /* First-In First-Out order        */
             QUE_CONVERT_ADDRESS,        /* Convert 16-bit addresses to 32  */
             szQueueName);               /* Name of the queue to create     */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosOpenQueue (&pidOwner,         /* PID of queue owner              */
                      &hqSpecialQue,     /* Handle for created queue        */
                      szQueueName);      /* Name of the queue to open       */
   if (rc!= NO_ERROR) {
      printf ("DosOpenQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "To be, or not to be.  That is the question...";
```

```
   rc = DosWriteQueue (hqSpecialQue,        /* Queue to write to           */
                       12345L,              /* Request data                */
                       strlen(DataBuffer),  /* Length of data to write     */
                       DataBuffer,          /* Pointer to data             */
                       0L);              /* Priority (not applicable here) */
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "";                  /* Clear the DataBuffer */
   Request.pid = pidOwner;           /* process ID for the DosReadQueue */
   rc = DosReadQueue (hqSpecialQue,          /* Queue to read from          */
                      &Request,              /* Request data from write     */
                      &ulDataLen,            /* Length of data returned     */
                      (PVOID) &DataBuffer,   /* The data                    */
                      0L,                    /* Remove first element        */
                      DCWW_WAIT,             /* Wait for available data     */
                      &ElemPrty,             /* Priority of data (returned) */
                      0L);             /* Semaphore to use when not waiting */

   if (rc!= NO_ERROR) {
      printf ("DosReadQueue error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosReadQueue returns: '%s'\n", DataBuffer);
      printf ("      (Request data = %u)\n", Request.ulData);
   }

   rc = DosCloseQueue(hqSpecialQue);         /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosCloseQueue - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCloseVDD

DosCloseVDD - Syntax

Closes the specified virtual device driver (VDD) handle.

```
#define INCL_DOSMVDM
#include <os2.h>
```

```
HVDD      hvdd;  /*  The handle of the virtual device driver to be closed. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosCloseVDD(hvdd);
```

DosCloseVDD Parameter - hvdd

hvdd (HVDD) - input
The handle of the virtual device driver to be closed.

Specify the handle that was returned by a previous call to DosOpenVDD.

DosCloseVDD Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCloseVDD returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
644	ERROR_INVALID_CALLER

For a full list of error codes, see Errors.

DosCloseVDD - Parameters

hvdd (HVDD) - input
The handle of the virtual device driver to be closed.

Specify the handle that was returned by a previous call to DosOpenVDD.

ulrc (APIRET) - returns
Return Code.

DosCloseVDD returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
644	ERROR_INVALID_CALLER

For a full list of error codes, see Errors.

DosCloseVDD - Remarks

DosCloseVDD closes the specified virtual device driver (VDD) handle.

DosCloseVDD - Related Functions

Related Functions

- DosCloseVDD
- DosOpenVDD
- DosRequestVDD

DosCloseVDD - Example Code

The following is NOT a complete C program. It is simply intended to provide an idea of how a protected-mode OS/2 process can communicate with a virtual device driver (VDD).

This example shows a protected-mode process calling a hypothetical VDD with a request to read a string of bytes from the VDD. Assume that the session identifier of the specified DOS session has been placed into SessionID already and that the sample virtual device driver has registered the name "VDD007" with the operating system.

```
#define INCL_DOSMVDM     /* Multiple DOS sessions values */
#define INCL_DOSERRORS   /* DOS Error values */
#include <os2.h>
#include <stdio.h>

UCHAR   VDDName[10] = "VDD007";    /* Name of VDD */
HVDD    VDDHandle   = NULLHANDLE;  /* Handle of VDD */
SGID    SessionID   = 0;  /* Session identifier (should be initialized */
ULONG   Command     = 3;           /* VDD function code (hypothetical) */
APIRET  rc          = NO_ERROR;    /* Return code */
UCHAR   InputBuffer[30]  = "Your command here";  /* Command information    */
UCHAR   OutputBuffer[30] = "";                   /* Output data (returned) */

   rc = DosOpenVDD(VDDName, &VDDHandle);
   if (rc != NO_ERROR) {
      printf("VDD %s was not found.\n", rc);
      return 1;
   }

   rc = DosRequestVDD(VDDHandle,              /* Handle of VDD */
                      SessionID,              /* Session ID */
                      Command,                /* Command to send to VDD */
                      sizeof(InputBuffer),    /* Length of input */
                      InputBuffer,            /* Input buffer */
                      sizeof(OutputBuffer),   /* Length of output area */
                      OutputBuffer);          /* Output from command */
   if (rc != NO_ERROR) {
       printf("DosRequestVDD error: return code = %u\n", rc);
       return 1;
   }

   rc = DosCloseVDD(VDDHandle);               /* Close the VDD */
   if (rc != NO_ERROR) {
       printf("DosCloseVDD error: return code = %u\n", rc);
       return 1;
   }
```

DosCloseVDD - Topics

Select an item:
Syntax
Parameters
Returns
Remarks

Example Code
Related Functions
Glossary

DosConnectNPipe

DosConnectNPipe - Syntax

Prepares a named pipe for a client process.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE     hpipe;  /*  The named-pipe handle to connect (returned to the server process by DosCreateNPipe). */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosConnectNPipe(hpipe);
```

DosConnectNPipe Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to connect (returned to the server process by DosCreateNPipe).

DosConnectNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosConnectNPipe returns one of the following values:

0	NO_ERROR
95	ERROR_INTERRUPT
109	ERROR_BROKEN_PIPE
230	ERROR_BAD_PIPE
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosConnectNPipe - Parameters

hpipe (HPIPE) - input
The named-pipe handle to connect (returned to the server process by DosCreateNPipe).

ulrc (APIRET) - returns
Return Code.

DosConnectNPipe returns one of the following values:

0	NO_ERROR
95	ERROR_INTERRUPT
109	ERROR_BROKEN_PIPE
230	ERROR_BAD_PIPE
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosConnectNPipe - Remarks

DosConnectNPipe is issued by a server process to put a named pipe into the listening state. This enables a client process to gain access to the pipe by calling DosOpen.

If the client end of the pipe is already open when DosConnectNPipe is issued, DosConnectNPipe returns immediately and has no effect. If the client end is closed, the result depends on whether the pipe is in blocking mode or nonblocking mode. (Blocking/nonblocking mode is specified when the pipe is created; it can also be changed by DosSetNPHState).

- If the pipe is in blocking mode, DosConnectNPipe waits for a client to open the pipe before returning.
- If the pipe is in nonblocking mode, DosConnectNPipe returns immediately with ERROR_PIPE_NOT_CONNECTED. Nevertheless, the pipe is placed into the listening state, permitting a client to subsequently issue DosOpen successfully.

Multiple DosConnectNPipe calls can be issued for a pipe that is in nonblocking mode. If the pipe is not already either open or closing, the first call to DosConnectNPipe puts the pipe into the listening state; subsequent calls merely test the pipe state.

If the pipe was previously opened and then closed by a client, but has not yet been disconnected by the server, DosConnectNPipe returns ERROR_BROKEN_PIPE.

If the function is interrupted while it is waiting for a client to open the pipe, ERROR_INTERRUPT is returned.

If DosConnectNPipe is called by a client process, ERROR_BAD_PIPE is returned.

Also, ERROR_BAD_PIPE is returned if you specify an invalid name or file handle.

DosConnectNPipe - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosConnectNPipe - Example Code

This example handles host end of a named pipe for several other named pipe examples. Some return code checking has been omitted. Compile and run this program before running client programs to the named pipe.

```
#define INCL_BASE
#define INCL_DOSSEMAPHORES
#define INCL_DOSNMPIPES
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
CHAR     PipeName[256] = "\\PIPE\\EXAMPLE" ;     /* Pipe name */
HPIPE    PipeHandle    = NULLHANDLE;             /* Pipe handle */
HEV      hev           = NULLHANDLE;             /* Semaphore handle */
ULONG    ulBytes       = 0;                      /* Bytes read or written */
CHAR     message[256]  = "";                     /* Input/Output buffer */
APIRET   rc            = NO_ERROR;               /* Return code */

   rc = DosCreateNPipe(PipeName,               /* Name of pipe to create */
                       &PipeHandle,            /* Handle returned for pipe */
                       NP_ACCESS_DUPLEX,       /* Duplex pipe */
                       NP_WAIT |
                       NP_TYPE_MESSAGE |
                       NP_READMODE_MESSAGE |
                       NP_WMESG |              /* Write messages */
                       NP_RMESG |              /* Read messages */
                       0x01,                   /* Unique instance of pipe */
                       sizeof(message),        /* Output buffer size */
                       sizeof(message),        /* Input buffer size */
                       0L);                    /* Use default time-out */
   if (rc != NO_ERROR) {
     printf("DosCreateNPipe error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCreateEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev, 0L, 0L);
   /* Should check if (rc != NO_ERROR)  here... This semaphore is not
      always used.                                                    */

   rc = DosSetNPipeSem(PipeHandle,       /* Handle for pipe */
                       (HSEM) hev,       /* Handle of semaphore */
                       1L);              /* Used to distinguish among events */
   if (rc != NO_ERROR) {
     printf("DosSetNPipeSem error: return code = %u\n",rc);
     return 1;
   }

   printf("Waiting for connection to pipe %s...\n",PipeName);

   rc = DosConnectNPipe(PipeHandle);
   if (rc != NO_ERROR) {
     printf("DosConnectNPipe error: return code = %u\n",rc);
     return 1;
   }

   printf("\nCONNECTED\nWaiting for a message...\n");
   rc = DosRead(PipeHandle,            /* Handle of pipe */
                message,               /* Buffer for message read */
                sizeof(message),       /* Buffer size */
                &ulBytes);             /* Number of bytes actually read */
   if (rc != NO_ERROR) {
     printf("DosRead error: return code = %u\n",rc);
     return 1;
   }

   printf("\n\nMessage received was: %s\n\n", message);

   strcpy(message, "Thank you for your message!");
   rc = DosWrite(PipeHandle,           /* Handle of pipe */
                 message,              /* Buffer containing message to write */
                 strlen(message),      /* Length of message */
                 &ulBytes);            /* Number of bytes actually written */
   if (rc != NO_ERROR) {
     printf("DosWrite error: return code = %u\n",rc);
```

```
      return 1;
   }

   rc = DosCloseEventSem(hev);
   /* Should check if (rc != NO_ERROR)  here... */

   rc = DosDisConnectNPipe(PipeHandle);
   /* Should check if (rc != NO_ERROR)  here... */

   return NO_ERROR;
}
```

DosConnectNPipe - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCopy

DosCopy - Syntax

Copies the source file or subdirectory to the destination file or subdirectory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszOld;  /*  Address of the ASCIIZ path name of the source file, subdirectory, or character device. */
PSZ       pszNew;  /*  Address of the ASCIIZ path name of the target file, subdirectory, or character device. */
ULONG     option;  /*  ULONG bit flags that define how the DosCopy function is done. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosCopy(pszOld, pszNew, option);
```

DosCopy Parameter - pszOld

pszOld (PSZ) - input
Address of the ASCIIZ path name of the source file, subdirectory, or character device.

Global file-name characters are not allowed.

DosCopy Parameter - pszNew

pszNew (PSZ) - input
Address of the ASCIIZ path name of the target file, subdirectory, or character device.

Global file-name characters are not allowed.

DosCopy Parameter - option

option (ULONG) - input
ULONG bit flags that define how the DosCopy function is done.

Bit	Description
31-3	Reserved. These bits must be set to zero.
2	DCPY_FAILEAS (0x00000004) Discard the EAs if the source file contains EAs and the destination file system does not support EAs. 0 Discard the EAs (extended attributes) if the destination file system does not support EAs. 1 Fail the copy if the destination file system does not support EAs.
1	DCPY_APPEND (x00000002) Append the source file to the target file's end of data. 0 Replace the target file with the source file. 1 Append the source file to the target file's end of data. This is ignored when copying a directory, or if the target file does not exist.
0	DCPY_EXISTING (0x00000001) Existing Target File Disposition. 0 Do not copy the source file to the target if the file name already exists within the target directory. If a single file is being copied and the target already exists, an error is returned. 1 Copy the source file to the target even if the file name already exists within the target directory.

Bit flag DCPY_FAILEAS can be used in combination with bit flag DCPY_APPEND or DCPY_EXISTING.

DosCopy Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCopy returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK

32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
267	ERROR_DIRECTORY
282	ERROR_EAS_NOT_SUPPORTED
283	ERROR_NEED_EAS_FOUND

For a full list of error codes, see Errors.

DosCopy - Parameters

pszOld (PSZ) - input
Address of the ASCIIZ path name of the source file, subdirectory, or character device.

Global file-name characters are not allowed.

pszNew (PSZ) - input
Address of the ASCIIZ path name of the target file, subdirectory, or character device.

Global file-name characters are not allowed.

option (ULONG) - input
ULONG bit flags that define how the DosCopy function is done.

Bit	Description
31-3	Reserved. These bits must be set to zero.
2	DCPY_FAILEAS (0x00000004) Discard the EAs if the source file contains EAs and the destination file system does not support EAs. 0 Discard the EAs (extended attributes) if the destination file system does not support EAs. 1 Fail the copy if the destination file system does not support EAs.
1	DCPY_APPEND (x00000002) Append the source file to the target file's end of data. 0 Replace the target file with the source file. 1 Append the source file to the target file's end of data. This is ignored when copying a directory, or if the target file does not exist.
0	DCPY_EXISTING (0x00000001) Existing Target File Disposition. 0 Do not copy the source file to the target if the file name already exists within the target directory. If a single file is being copied and the target already exists, an error is returned. 1 Copy the source file to the target even if the file name already exists within the target directory.

Bit flag DCPY_FAILEAS can be used in combination with bit flag DCPY_APPEND or DCPY_EXISTING.

ulrc (APIRET) - returns
Return Code.

DosCopy returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED

26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
267	ERROR_DIRECTORY
282	ERROR_EAS_NOT_SUPPORTED
283	ERROR_NEED_EAS_FOUND

For a full list of error codes, see Errors.

DosCopy - Remarks

DosCopy copies all files and subdirectories in the source path to the target path. Global file-name characters are not allowed in source or target names. The source and the target can be on different drives.

If an I/O error occurs, DosCopy takes the following actions:

- If the source name is that of a subdirectory, deletes the file being copied from the target path.
- If the source name is that of a file to be replaced, deletes the file from the target path.
- If the source name is that of a file to be appended, resizes the target file to its original size.

Read-only files in the target path cannot be replaced by a DosCopy request. If such files exist in the target, and *option* bit flag DCPY_EXISTING is set to 1, any attempt to replace these files with files from the source will result in an error.

When copying is specified for a single file that has *option* bit flag DCPY_APPEND set to 1, the operation proceeds even if the file already exists and its *option* bit flag DCPY_EXISTING is set to 0. That is, *option* bit flag DCPY_EXISTING is significant only when replacing a file, not when appending a file.

If a device name is specified as the target, the source name must be a file, not a directory. When the request is issued, *option* bit flags DCPY_EXISTING and DCPY_APPEND are ignored.

File-object attributes, such as date of creation, and time of creation, are always copied from the source to the target; however, extended attributes (EAs) are not copied in every case. DosCopy copies EAs from the source to the target when creating a file or directory, or when replacing an existing file on the target; however, it does not copy them when appending an existing file or when copying files to an existing directory on the target. If the file system of the target does not support EAs, DosCopy ends and returns an error.

If the source file object contains a needed EA, and the destination file system does not support EAs, DosCopy fails regardless of the value of *option* bit flag DCPY_FAILEAS.

DosQuerySysInfo should be called by an application during initialization to determine the maximum path length allowed by the operating system.

DosCopy - Related Functions

Related Functions

- DosMove
- DosQueryCurrentDisk
- DosQuerySysInfo
- DosSetDefaultDisk

DosCopy - Example Code

This example creates a backup copy of the file "CONFIG.SYS" with the new name "CONFIG.CPY", in the root directory, even if the new file

name already exists.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>
 #include <string.h>

int main(VOID) {
   UCHAR   achSourceString[80]  = "config.sys"; /* String to transform */
   UCHAR   achEditString[80]    = "*.cpy";      /* Editing string */
   UCHAR   achTargetString[200] = "";           /* Destination string buffer */
   APIRET  rc                   = NO_ERROR;     /* Return code */

   rc = DosSetDefaultDisk(3);    /* Set drive to C: (1=A, 2=B, 3=C, ...)      */
   if (rc != NO_ERROR) {
      printf("DosSetDefaultDisk error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir ("\\");  /*  Set directory to root */
   if (rc != NO_ERROR) {
      printf("DosSetCurrentDir error: return code = %u\n", rc);
      return 1;
   }

           /* Transform "CONFIG.SYS" using "*.CPY" to "CONFIG.CPY" */

   rc = DosEditName(1, achSourceString, achEditString, achTargetString, 200);

   if (rc != NO_ERROR) {
      printf("DosEditName error: return code = %u\n", rc);
      return 1;
   }

           /* Copy contents of CONFIG.SYS to the backup file */

   rc = DosCopy(achSourceString,             /* Name of file to be copied    */
                achTargetString,             /* Name of the target file      */
                DCPY_EXISTING);   /* Copy even if target file already exists */

   if (rc != NO_ERROR) {
      printf("DosCopy error: return code = %u\n", rc);
      return 1;
   } else  printf ("Backup file %s created.\n", achTargetString);
   return NO_ERROR;
}
```

DosCopy - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCreateDir

DosCreateDir - Syntax

Creates a new directory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszDirName;  /*  Address of the ASCIIZ directory path name, which may contain a drive specification. *
PEAOP2    peaop2;      /*  Address of the extended attribute buffer, which contains an EAOP2 data structure. */
APIRET    ulrc;        /*  Return Code. */

ulrc = DosCreateDir(pszDirName, peaop2);
```

DosCreateDir Parameter - pszDirName

pszDirName (PSZ) - input
Address of the ASCIIZ directory path name, which may contain a drive specification.

If no drive is specified, the current drive is assumed.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

DosCreateDir Parameter - peaop2

peaop2 (PEAOP2) - in/out
Address of the extended attribute buffer, which contains an EAOP2 data structure.

On input, the *fpGEA2List* field and *oError* fields are ignored. The EA setting operation is performed on the information contained in *fpFEA2List*. If extended attributes are not to be defined or modified, then *peaop2* must be set to null.

On output, *fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the directory.

DosCreateDir Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateDir returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

ERROR_EA_VALUE_UNSUPPORTABLE

For a full list of error codes, see Errors.

DosCreateDir - Parameters

pszDirName (PSZ) - input
Address of the ASCIIZ directory path name, which may contain a drive specification.

If no drive is specified, the current drive is assumed.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

peaop2 (PEAOP2) - in/out
Address of the extended attribute buffer, which contains an EAOP2 data structure.

On input, the *fpGEA2List* field and *oError* fields are ignored. The EA setting operation is performed on the information contained in *fpFEA2List*. If extended attributes are not to be defined or modified, then *peaop2* must be set to null.

On output, *fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the directory.

ulrc (APIRET) - returns
Return Code.

DosCreateDir returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT
	ERROR_EA_VALUE_UNSUPPORTABLE

For a full list of error codes, see Errors.

DosCreateDir - Remarks

DosCreateDir enables an application to define extended attributes for a subdirectory at the time of its creation.

If any subdirectory names specified in the path name do not exist, the subdirectory is not created. Upon successful return, a subdirectory is created at the end of the specified path.

An application must issue DosQuerySysInfo to determine the maximum path length that the operating system supports. The returned value should be used to dynamically allocate buffers that are to be used to store paths.

If a program with its NEWFILES bit set tries to create a directory on an FAT file system drive and specifies blanks immediately preceding the dot in a file name, the system rejects the name. For example, if c: is an FAT file system drive, the name "file .txt" is rejected, but "file.txt" is accepted.

DosCreateDir - Related Functions

Related Functions

- DosQuerySysInfo

DosCreateDir - Example Code

This example creates the new directory "TEMPPROG" in the root directory, access it, and then deletes it. Some return codes are not checked for brevity.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   UCHAR   achOrigDirName[256] = "";            /* Original directory name  */
   UCHAR   achNewDirName[256]  = "\\TEMPPROG";  /* New directory to create  */
   UCHAR   achDirName[256]     = "";          /* Directory name for queries */
   ULONG   cbDirPathLen    = 0;               /* Length of directory path   */
   PEAOP2  pEABuf       = NULL;   /* Extended Attribute buffer pointer      */
   ULONG   ulDriveNum   = 0;      /* Drive number: current=0, A=1, B=2, ... */
   APIRET  rc           = NO_ERROR;      /* Return code                      */

   cbDirPathLen = (ULONG) sizeof(achOrigDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achOrigDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Original dir. = \\%s\n", achOrigDirName);

   pEABuf = NULL;  /* Indicate no EAs are to be defined for the directory  */
   rc = DosCreateDir(achNewDirName, pEABuf);   /* Create the new directory */

   if (rc != NO_ERROR) {
      printf("DosCreateDir error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir (achNewDirName);   /* Change to new directory     */

   ulDriveNum = 0;
   cbDirPathLen = (ULONG) sizeof(achDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Current dir.  = \\%s\n", achDirName);

   strcpy(achDirName,"\\");

   rc = DosSetCurrentDir (achDirName);     /* Change to root directory     */
   rc = DosDeleteDir (achNewDirName);      /* Delete the new directory     */

   return NO_ERROR;
}
```

DosCreateDir - Topics

Select an item:
Syntax
Parameters

Returns
Remarks
Example Code
Related Functions
Glossary

DosCreateEventSem

DosCreateEventSem - Syntax

Creates an event semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

PSZ       pszName;  /*  A pointer to the ASCIIZ name of the semaphore. */
PHEV      phev;     /*  A pointer to the handle of the event semaphore. */
ULONG     flAttr;   /*  A set of flags that specify the attributes of the event semaphore. */
BOOL32    fState;   /*  Initial state of the semaphore. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosCreateEventSem(pszName, phev, flAttr,
         fState);
```

DosCreateEventSem Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the semaphore is unnamed. Unnamed event semaphores can be either private or shared, depending on *flAttr*. They are identified by the semaphore handle that *phev* points to.

By default, all named semaphores are shared.

DosCreateEventSem Parameter - phev

phev (PHEV) - output
A pointer to the handle of the event semaphore.

DosCreateEventSem Parameter - flAttr

flAttr (ULONG) - input
A set of flags that specify the attributes of the event semaphore.

If the DC_SEM_SHARED bit is set, the semaphore is shared. Otherwise, this flag should be set to 0L. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.

DosCreateEventSem Parameter - fState

fState (BOOL32) - input
Initial state of the semaphore.

Possible values are defined in the list below:

0	FALSE The initial state of the semaphore is "reset."
1	TRUE The initial state of the semaphore is "posted."

DosCreateEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateEventSem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES

For a full list of error codes, see Errors.

DosCreateEventSem - Parameters

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the semaphore is unnamed. Unnamed event semaphores can be either private or shared, depending on *flAttr*. They are identified by the semaphore handle that *phev* points to.

By default, all named semaphores are shared.

phev (PHEV) - output
A pointer to the handle of the event semaphore.

flAttr (ULONG) - input
A set of flags that specify the attributes of the event semaphore.

If the DC_SEM_SHARED bit is set, the semaphore is shared. Otherwise, this flag should be set to 0L. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.

fState (BOOL32) - input
Initial state of the semaphore.

Possible values are defined in the list below:

0	FALSE The initial state of the semaphore is "reset."
1	TRUE The initial state of the semaphore is "posted."

ulrc (APIRET) - returns
Return Code.

DosCreateEventSem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES

For a full list of error codes, see Errors.

DosCreateEventSem - Related Functions

Related Functions

- DosCloseEventSem
- DosOpenEventSem
- DosPostEventSem
- DosQueryEventSem
- DosResetEventSem
- DosWaitEventSem

DosCreateEventSem - Example Code

This example creates an event semaphore, and the asyncronous timer posts to it when its time interval expires. Finally, the event semaphore is closed.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

PSZ     szSemName  = "\\SEM32\\TIMER\\THREAD1\\EVENT1"; /* Semaphore name */
HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
APIRET  rc            = NO_ERROR;           /* Return code               */

   rc = DosCreateEventSem(szSemName,      /* Name of semaphore to create  */
                          &hevEvent1,     /* Handle of semaphore returned */
                          DC_SEM_SHARED,  /* Shared semaphore             */
                          FALSE);         /* Semaphore is in RESET state  */
   if (rc != NO_ERROR) {
```

```
        printf("DosCreateEventSem error: return code = %u\n", rc);
        return 1;       }

    rc = DosAsyncTimer(7000L,              /* 7 second interval            */
                       (HSEM) hevEvent1,   /* Semaphore to post            */
                       &htimerEvent1);     /* Timer handler (returned)     */
    if (rc != NO_ERROR) {
        printf("DosAsyncTimer error: return code = %u\n", rc);
        return 1;
    } else {
        printf("Timer will expire in about 7 seconds...\n");
    }

           /* ... add your other processing here... */

    rc = DosWaitEventSem(hevEvent1,            /* Wait for AsyncTimer event */
                (ULONG) SEM_INDEFINITE_WAIT); /* As long as it takes       */
    if (rc != NO_ERROR) {
        printf("DosWaitEventSem error: return code = %u\n", rc);
        return 1;
    }

    rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u", rc);
        return 1;
    }

return NO_ERROR;
}
```

DosCreateEventSem - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosCreateMutexSem

DosCreateMutexSem - Syntax

Creates a mutex semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

PSZ       pszName;  /*  Pointer to the ASCIIZ name of the semaphore. */
PHMTX     phmtx;    /*  A pointer to the handle of the mutex semaphore. */
ULONG     flAttr;   /*  A set of flags that specify the attributes of the semaphore. */
BOOL32    fState;   /*  Initial state of the semaphore. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosCreateMutexSem(pszName, phmtx, flAttr,
         fState);
```

DosCreateMutexSem Parameter - pszName

pszName (PSZ) - input
Pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the semaphore is unnamed. Unnamed mutex semaphores can be either private or shared, depending on *flAttr* They are identified by the semaphore handle that *phmtx* points to.

By default, named semaphores are shared.

DosCreateMutexSem Parameter - phmtx

phmtx (PHMTX) - output
A pointer to the handle of the mutex semaphore.

DosCreateMutexSem Parameter - flAttr

flAttr (ULONG) - input
A set of flags that specify the attributes of the semaphore.

If the DC_SEM_SHARED bit is set, the semaphore is shared. Otherwise, this flag should be set to 0L. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.

DosCreateMutexSem Parameter - fState

fState (BOOL32) - input
Initial state of the semaphore.

Possible values are shown in the following list:

0	FALSE The initial state of the semaphore is "unowned."
1	TRUE The initial state of the semaphore is "owned."

DosCreateMutexSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateMutexSem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES

For a full list of error codes, see Errors.

DosCreateMutexSem - Parameters

pszName (PSZ) - input
Pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the semaphore is unnamed. Unnamed mutex semaphores can be either private or shared, depending on *flAttr* They are identified by the semaphore handle that *phmtx* points to.

By default, named semaphores are shared.

phmtx (PHMTX) - output
A pointer to the handle of the mutex semaphore.

flAttr (ULONG) - input
A set of flags that specify the attributes of the semaphore.

If the DC_SEM_SHARED bit is set, the semaphore is shared. Otherwise, this flag should be set to 0L. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.

fState (BOOL32) - input
Initial state of the semaphore.

Possible values are shown in the following list:

0	FALSE The initial state of the semaphore is "unowned."
1	TRUE The initial state of the semaphore is "owned."

ulrc (APIRET) - returns
Return Code.

DosCreateMutexSem returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES

For a full list of error codes, see Errors.

DosCreateMutexSem - Remarks

DosCreateMutexSem creates a mutual exclusion (mutex) semaphore, and opens it for all of the threads in the current process.

DosCreateMutexSem - Related Functions

Related Functions

- DosCloseMutexSem
- DosOpenMutexSem
- DosQueryMutexSem
- DosReleaseMutexSem
- DosRequestMutexSem

DosCreateMutexSem - Example Code

This example creates a named mutual exclusion (Mutex) semaphore, manipulates it, and finally closes it.

```
#define INCL_DOSSEMAPHORES      /* Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HMTX    hmtx     = NULLHANDLE; /* Mutex semaphore handle */
PID     pidOwner = 0;          /* PID of current mutex semaphore owner */
TID     tidOwner = 0;          /* TID of current mutex semaphore owner */
ULONG   ulCount  = 0;          /* Request count for the semaphore */
APIRET  rc       = NO_ERROR;   /* Return code */

   rc = DosCreateMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                          &hmtx, 0, FALSE);       /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }
   /* This would normally be done by another unit of work */
   rc = DosOpenMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                        &hmtx);                 /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosRequestMutexSem(hmtx,                     /* Handle of semaphore */
                           (ULONG) SEM_INDEFINITE_WAIT);  /* Timeout (none) */
   if (rc != NO_ERROR) {
      printf("DosRequestMutexSem error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryMutexSem(hmtx,         /* Handle of semaphore */
                         &pidOwner,    /* Process ID of owner */
                         &tidOwner,    /* Thread ID of owner */
                         &ulCount);    /* Count */
   if (rc != NO_ERROR) {
      printf("DosQueryMutexSem error: return code = %u\n", rc);
      return 1;
   } else  {
      printf("Semaphore owned by PID %u, TID %u.", pidOwner, tidOwner);
      printf("  Request count is %u.\n", ulCount);
   } /* endif */

   rc = DosReleaseMutexSem(hmtx);        /* Relinquish ownership */
   if (rc != NO_ERROR) {
      printf("DosReleaseMutexSem error: return code = %u\n", rc);
      return 1;
    }
```

```
   rc = DosCloseMutexSem(hmtx);          /* Close mutex semaphore */
   if (rc != NO_ERROR) {
      printf("DosCloseMutexSem error: return code = %u\n", rc);
      return 1;
   }

return NO_ERROR;
}
```

DosCreateMutexSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCreateMuxWaitSem

DosCreateMuxWaitSem - Syntax

Creates a multiple wait (muxwait) semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

PSZ           pszName;  /*  A pointer to the ASCIIZ name of the semaphore. */
PHMUX         phmux;    /*  A pointer to the handle of the muxwait semaphore. */
ULONG         cSemRec;  /*  The count of semaphore record entries in pSemRec. */
PSEMRECORD    pSemRec;  /*  A pointer to the array of semaphore record entries to put into the muxwait semaphore
ULONG         flAttr;   /*  A set of flags that specify the attributes of the semaphore. */
APIRET        ulrc;     /*  Return Code. */

ulrc = DosCreateMuxWaitSem(pszName, phmux,
         cSemRec, pSemRec, flAttr);
```

DosCreateMuxWaitSem Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the muxwait semaphore is unnamed. Unnamed semaphores can be either private or shared, depending on *flAttr*. They are identified by the semaphore handle that *phmux* points to.

By default, named semaphores are shared.

DosCreateMuxWaitSem Parameter - phmux

phmux (PHMUX) - output
A pointer to the handle of the muxwait semaphore.

DosCreateMuxWaitSem Parameter - cSemRec

cSemRec (ULONG) - input
The count of semaphore record entries in *pSemRec*.

DosCreateMuxWaitSem Parameter - pSemRec

pSemRec (PSEMRECORD) - input
A pointer to the array of semaphore record entries to put into the muxwait semaphore.

This is the list of event or mutex semaphores that must be posted or released in order for the muxwait semaphore to clear.

DosCreateMuxWaitSem Parameter - flAttr

flAttr (ULONG) - input
A set of flags that specify the attributes of the semaphore.

The attributes of the semaphore are a combination of the following values:

1 DC_SEM_SHARED
The semaphore is shared when this bit is set; otherwise, it is a private semaphore. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.

2 DCMW_WAIT_ANY.
The semaphore clears when any event semaphore in its *pSemRec* list is posted, or when any mutex semaphore in its *pSemRec* list is released. When any one of the semaphores is cleared, the thread waiting for the muxwait semaphore can continue execution.

4 DCMW_WAIT_ALL
The semaphore clears when all of the event semaphores in its *pSemRec* list have been posted, or when all of the mutex semaphores in its *pSemRec* list have been released. When all of the semaphores are cleared, the thread waiting for the muxwait semaphore can continue execution.

DosCreateMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
100	ERROR_TOO_MANY_SEMAPHORES
105	ERROR_SEM_OWNER_DIED
123	ERROR_INVALID_NAME
284	ERROR_DUPLICATE_HANDLE
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES
292	ERROR_WRONG_TYPE

For a full list of error codes, see Errors.

DosCreateMuxWaitSem - Parameters

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the muxwait semaphore is unnamed. Unnamed semaphores can be either private or shared, depending on *flAttr*. They are identified by the semaphore handle that *phmux* points to.

By default, named semaphores are shared.

phmux (PHMUX) - output
A pointer to the handle of the muxwait semaphore.

cSemRec (ULONG) - input
The count of semaphore record entries in *pSemRec*.

pSemRec (PSEMRECORD) - input
A pointer to the array of semaphore record entries to put into the muxwait semaphore.

This is the list of event or mutex semaphores that must be posted or released in order for the muxwait semaphore to clear.

flAttr (ULONG) - input
A set of flags that specify the attributes of the semaphore.

The attributes of the semaphore are a combination of the following values:

1	DC_SEM_SHARED The semaphore is shared when this bit is set; otherwise, it is a private semaphore. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.
2	DCMW_WAIT_ANY. The semaphore clears when any event semaphore in its *pSemRec* list is posted, or when any mutex semaphore in its *pSemRec* list is released. When any one of the semaphores is cleared, the thread waiting for the muxwait semaphore can continue execution.
4	DCMW_WAIT_ALL The semaphore clears when all of the event semaphores in its *pSemRec* list have been posted, or when all of the mutex semaphores in its *pSemRec* list have been released. When all of the semaphores are cleared, the thread waiting for the muxwait semaphore can continue execution.

ulrc (APIRET) - returns
Return Code.

DosCreateMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
100	ERROR_TOO_MANY_SEMAPHORES
105	ERROR_SEM_OWNER_DIED
123	ERROR_INVALID_NAME
284	ERROR_DUPLICATE_HANDLE
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES
292	ERROR_WRONG_TYPE

For a full list of error codes, see Errors.

DosCreateMuxWaitSem - Remarks

DosCreateMuxWaitSem creates a multiple wait (muxwait) semaphore, and opens it for all of the threads in the current process.

DosCreateMuxWaitSem - Related Functions

Related Functions

- DosAddMuxWaitSem
- DosCloseMuxWaitSem
- DosDeleteMuxWaitSem
- DosOpenMuxWaitSem
- DosQueryMuxWaitSem
- DosWaitMuxWaitSem

DosCreateMuxWaitSem - Example Code

This example creates a semaphore record list with five event semaphores. It then creates a MuxWait semaphore with the semaphore record list in its list, waits, deletes the first event semaphore from its list, and closes the MuxWait semaphore.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX      hmuxHandAny = NULLHANDLE;      /* Muxwaithandle */
 HEV       hev[5]      = {0};             /* Event semaphores */
 SEMRECORD apsr[5]     = {{0}};           /* Semaphore records */
 APIRET    rc          = NO_ERROR;        /* Return code */
 ULONG     ulLoop      = 0;               /* Loop count */
 ULONG     ulSem       = 0;

for (ulLoop = 0; ulLoop < 5; ulLoop++) {
    rc = DosCreateEventSem((PSZ) NULL,
                           &hev[ulLoop],
                           0,
                           FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
    apsr[ulLoop].hsemCur = (HSEM) hev[ulLoop],
    apsr[ulLoop].ulUser = 0;
```

```
} /* endfor */

rc = DosCreateMuxWaitSem((PSZ) NULL,
                         &hmuxHandAny,
                         5L,              /* Number of semaphores in list */
                         apsr,            /* Semaphore list */
                         DCMW_WAIT_ANY);  /* Wait for any semaphore */
    if (rc != NO_ERROR) {
      printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
      return 1;
      }

rc = DosWaitMuxWaitSem(hmuxHandAny,
                       SEM_IMMEDIATE_RETURN,
                       &ulSem);       /* No semaphores have been posted, so
                                         we should see a timeout below...    */
    if (rc != ERROR_TIMEOUT) {
      printf("DosWaitMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosDeleteMuxWaitSem(hmuxHandAny, apsr[0].hsemCur);
    if (rc != NO_ERROR) {
      printf("DosDeleteMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }
rc = DosCloseMuxWaitSem(hmuxHandAny);
     if (rc != NO_ERROR) {
       printf("DosCloseMuxWaitSem error: return code = %u\n", rc);
       return 1;
     }

return NO_ERROR;
}
```

DosCreateMuxWaitSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosCreateNPipe

DosCreateNPipe - Syntax

Creates a named pipe.

```
#define INCL_DOSNMPIPES
#include <os2.h>

PSZ       pszName;   /*  The ASCIIZ name of the pipe to be opened. */
PHPIPE    pHpipe;    /*  A pointer to the variable in which the system returns the handle of the pipe that is cr
ULONG     openmode;  /*  A set of flags defining the mode in which to open the pipe. */
```

```
ULONG     pipemode;  /*  A set of flags defining the mode of the pipe. */
ULONG     cbOutbuf;  /*  The number of bytes to allocate for the outbound (server to client) buffer. */
ULONG     cbInbuf;   /*  The number of bytes to allocate for the inbound (client to server) buffer. */
ULONG     msec;      /*  The maximum time, in milliseconds, to wait for a named-pipe instance to become availabl
APIRET    ulrc;      /*  Return Code. */

ulrc = DosCreateNPipe(pszName, pHpipe, openmode,
         pipemode, cbOutbuf, cbInbuf, msec);
```

DosCreateNPipe Parameter - pszName

pszName (PSZ) - input
The ASCIIZ name of the pipe to be opened.

Pipe names must include the prefix \PIPE\ and must conform to file-system naming conventions.

DosCreateNPipe Parameter - pHpipe

pHpipe (PHPIPE) - output
A pointer to the variable in which the system returns the handle of the pipe that is created.

DosCreateNPipe Parameter - openmode

openmode (ULONG) - input
A set of flags defining the mode in which to open the pipe.

This parameter contains the following bit fields:

Bit	Description
31-16	Reserved.
15	Reserved and must be 0.
14	Write-through bit. Possible values are shown in the following list: 0 NP_WRITEBEHIND (0x0000) Write-behind to remote pipes is allowed. 1 NP_NOWRITEBEHIND (0x4000) Write-behind to remote pipes is not allowed. This bit is meaningful only for a remote pipe. Occasionally, data written to a remote pipe is buffered locally and then sent across the network to the pipe at a later time. Setting the write-through bit ensures that data is sent to the remote pipe as soon as it is written.
13-8	Reserved.
7	Inheritance flag. Possible values are shown in the following list: 0 NP_INHERIT (0x0000) The pipe handle is inherited by a child process. 1 NP_NOINHERIT (0x0080)

The pipe handle is private to the current process and cannot be inherited.

This bit is not inherited by child processes.

6-3 Reserved and must be 0.

2-0 Access mode. The pipe access is defined as shown in the following list:

000 NP_ACCESS_INBOUND (0x0000)
Inbound pipe (client to server).

001 NP_ACCESS_OUTBOUND (0x0001)
Outbound pipe (server to client).

010 NP_ACCESS_DUPLEX (0x0002)
Duplex pipe (server to and from client).

Any other value is invalid.

DosCreateNPipe Parameter - pipemode

pipemode (ULONG) - input
A set of flags defining the mode of the pipe.

This parameter contains the following bit fields:

Bit	Description
31-16	Reserved.
15	Blocking mode. Blocking mode is defined as either "blocking" or "nonblocking," as follows: 0 NP_WAIT (0x0000) Blocking mode: DosRead and DosWrite block if no data is available. 1 NP_NOWAIT = NP_NBLK (0x8000) Nonblocking mode: DosRead and DosWrite return immediately if no data is available. DosRead normally blocks until at least partial data can be returned. DosWrite blocks by default until all of the requested bytes have been written. Nonblocking mode changes this behavior as following: DosRead returns immediately with ERROR_NO_DATA if no data is available. DosWrite returns immediately with a value of 0 for *pcbActual* if there is not enough buffer space available in the pipe; otherwise, the entire data area is transferred.
14-12	Reserved.
11-10	Type of named pipe. The pipe type is defined as follows: 00 NP_TYPE_BYTE (0x0000) The pipe is a byte pipe; that is, data is written to the pipe as an undifferentiated stream of bytes. 01 NP_TYPE_MESSAGE = NP_WMESG (0x0400) The pipe is a message pipe; that is, data is written to the pipe as messages. The system records the length of each message in the first two bytes of the message, which are called the *message header* . A header of all zeroes is reserved, and zero-length messages are not allowed.
9-8	Read mode. The read mode is defined as follows: 00 NP_READMODE_BYTE (0x0000) Byte-read mode: Read the pipe as a byte stream. 01 NP_READMODE_MESSAGE = NP_RMESG (0x0100) Message-read mode: Read the pipe as a message stream.

Message pipes can be read as either byte streams or message streams, depending on the value of this bit. Byte pipes can be read only as byte streams.

7-0 ICount (Instance count). When the first instance of a named pipe is created, ICount specifies how many instances (including the first instance) may be created. Possible values are shown in the following list:

1	This is the only instance permitted (the pipe is unique).
1 < value < 255	The number of instances is limited to the value specified.
-1	NP_UNLIMITED_INSTANCES (0x00FF) The number of instances is unlimited.
0	Reserved value.

ICount is ignored when specifying any instance of a pipe other than the first one. Subsequent attempts to create a pipe instance fail if the maximum number of allowed instances already exists. When multiple instances are allowed, multiple clients can simultaneously open the same pipe name; they will receive handles to distinct pipe instances.

DosCreateNPipe Parameter - cbOutbuf

cbOutbuf (ULONG) - input
The number of bytes to allocate for the outbound (server to client) buffer.

DosCreateNPipe Parameter - cbInbuf

cbInbuf (ULONG) - input
The number of bytes to allocate for the inbound (client to server) buffer.

DosCreateNPipe Parameter - msec

msec (ULONG) - input
The maximum time, in milliseconds, to wait for a named-pipe instance to become available.

This is the default value for the *msec* parameter of DosWaitNPipe.

This value may be set only when the first instance of the pipe name is being created. If the value is 0, a system-wide default value (50 ms) is chosen.

DosCreateNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateNPipe returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
8	ERROR_NOT_ENOUGH_MEMORY
84	ERROR_OUT_OF_STRUCTURES
87	ERROR_INVALID_PARAMETER
231	ERROR_PIPE_BUSY

On the PowerPC platform, DosCreateNPipe fails with error code ERROR_PATH_NOT_FOUND when the pipe name contains more than one \ in the name. For example, the following pipe name would cause DosCreateNPipe to fail:

```
\pipe\\\\\longname.nam
```

For a full list of error codes, see Errors.

DosCreateNPipe - Parameters

pszName (PSZ) - input
The ASCIIZ name of the pipe to be opened.

Pipe names must include the prefix \PIPE\ and must conform to file-system naming conventions.

pHpipe (PHPIPE) - output
A pointer to the variable in which the system returns the handle of the pipe that is created.

openmode (ULONG) - input
A set of flags defining the mode in which to open the pipe.

This parameter contains the following bit fields:

Bit	Description
31-16	Reserved.
15	Reserved and must be 0.
14	Write-through bit. Possible values are shown in the following list: 0 NP_WRITEBEHIND (0x0000) Write-behind to remote pipes is allowed. 1 NP_NOWRITEBEHIND (0x4000) Write-behind to remote pipes is not allowed. This bit is meaningful only for a remote pipe. Occasionally, data written to a remote pipe is buffered locally and then sent across the network to the pipe at a later time. Setting the write-through bit ensures that data is sent to the remote pipe as soon as it is written.
13-8	Reserved.
7	Inheritance flag. Possible values are shown in the following list: 0 NP_INHERIT (0x0000) The pipe handle is inherited by a child process. 1 NP_NOINHERIT (0x0080) The pipe handle is private to the current process and cannot be inherited. This bit is not inherited by child processes.
6-3	Reserved and must be 0.
2-0	Access mode. The pipe access is defined as shown in the following list: 000 NP_ACCESS_INBOUND (0x0000) Inbound pipe (client to server).

001	NP_ACCESS_OUTBOUND (0x0001) Outbound pipe (server to client).	
010	NP_ACCESS_DUPLEX (0x0002) Duplex pipe (server to and from client).	

Any other value is invalid.

pipemode (ULONG) - input

A set of flags defining the mode of the pipe.

This parameter contains the following bit fields:

Bit	Description
31-16	Reserved.
15	Blocking mode. Blocking mode is defined as either "blocking" or "nonblocking," as follows: 0 — NP_WAIT (0x0000) Blocking mode: DosRead and DosWrite block if no data is available. 1 — NP_NOWAIT = NP_NBLK (0x8000) Nonblocking mode: DosRead and DosWrite return immediately if no data is available. DosRead normally blocks until at least partial data can be returned. DosWrite blocks by default until all of the requested bytes have been written. Nonblocking mode changes this behavior as following: DosRead returns immediately with ERROR_NO_DATA if no data is available. DosWrite returns immediately with a value of 0 for *pcbActual* if there is not enough buffer space available in the pipe; otherwise, the entire data area is transferred.
14-12	Reserved.
11-10	Type of named pipe. The pipe type is defined as follows: 00 — NP_TYPE_BYTE (0x0000) The pipe is a byte pipe; that is, data is written to the pipe as an undifferentiated stream of bytes. 01 — NP_TYPE_MESSAGE = NP_WMESG (0x0400) The pipe is a message pipe; that is, data is written to the pipe as messages. The system records the length of each message in the first two bytes of the message, which are called the *message header* . A header of all zeroes is reserved, and zero-length messages are not allowed.
9-8	Read mode. The read mode is defined as follows: 00 — NP_READMODE_BYTE (0x0000) Byte-read mode: Read the pipe as a byte stream. 01 — NP_READMODE_MESSAGE = NP_RMESG (0x0100) Message-read mode: Read the pipe as a message stream. Message pipes can be read as either byte streams or message streams, depending on the value of this bit. Byte pipes can be read only as byte streams.
7-0	ICount (Instance count). When the first instance of a named pipe is created, ICount specifies how many instances (including the first instance) may be created. Possible values are shown in the following list: 1 — This is the only instance permitted (the pipe is unique). 1 < value < 255 — The number of instances is limited to the value specified. -1 — NP_UNLIMITED_INSTANCES (0x00FF) The number of instances is unlimited. 0 — Reserved value. ICount is ignored when specifying any instance of a pipe other than the first one. Subsequent

attempts to create a pipe instance fail if the maximum number of allowed instances already exists. When multiple instances are allowed, multiple clients can simultaneously open the same pipe name; they will receive handles to distinct pipe instances.

cbOutbuf (ULONG) - input
The number of bytes to allocate for the outbound (server to client) buffer.

cbInbuf (ULONG) - input
The number of bytes to allocate for the inbound (client to server) buffer.

msec (ULONG) - input
The maximum time, in milliseconds, to wait for a named-pipe instance to become available.

This is the default value for the *msec* parameter of DosWaitNPipe.

This value may be set only when the first instance of the pipe name is being created. If the value is 0, a system-wide default value (50 ms) is chosen.

ulrc (APIRET) - returns
Return Code.

DosCreateNPipe returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
8	ERROR_NOT_ENOUGH_MEMORY
84	ERROR_OUT_OF_STRUCTURES
87	ERROR_INVALID_PARAMETER
231	ERROR_PIPE_BUSY

On the PowerPC platform, DosCreateNPipe fails with error code ERROR_PATH_NOT_FOUND when the pipe name contains more than one \ in the name. For example, the following pipe name would cause DosCreateNPipe to fail:

```
\pipe\\\\\longname.nam
```

For a full list of error codes, see Errors.

DosCreateNPipe - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosDupHandle
- DosDisConnectNPipe
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosCreateNPipe - Example Code

This example handles host end of a named pipe for several other named pipe examples. Some return code checking has been omitted for brevity.

```
#define INCL_BASE
#define INCL_DOSSEMAPHORES
#define INCL_DOSNMPIPES
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
CHAR     PipeName[256] = "\\PIPE\\EXAMPLE" ;     /* Pipe name */
HPIPE    PipeHandle    = NULLHANDLE;             /* Pipe handle */
HEV      hev           = NULLHANDLE;             /* Semaphore handle */
ULONG    ulBytes       = 0;                      /* Bytes read or written */
CHAR     message[256]  = "";                     /* Input/Output buffer */
APIRET   rc            = NO_ERROR;               /* Return code */

   rc = DosCreateNPipe(PipeName,                /* Name of pipe to create */
                       &PipeHandle,             /* Handle returned for pipe */
                       NP_ACCESS_DUPLEX,        /* Duplex pipe */
                       NP_WAIT |
                       NP_TYPE_MESSAGE |
                       NP_READMODE_MESSAGE |
                       NP_WMESG |               /* Write messages */
                       NP_RMESG |               /* Read messages */
                       0x01,                    /* Unique instance of pipe */
                       sizeof(message),         /* Output buffer size */
                       sizeof(message),         /* Input buffer size */
                       0L);                     /* Use default time-out */
   if (rc != NO_ERROR) {
     printf("DosCreateNPipe error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCreateEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev, 0L, 0L);
   /* Should check if (rc != NO_ERROR)  here... This semaphore is not
      always used.                                                    */

   rc = DosSetNPipeSem(PipeHandle,       /* Handle for pipe */
                       (HSEM) hev,       /* Handle of semaphore */
                       1L);              /* Used to distinguish among events */
   if (rc != NO_ERROR) {
     printf("DosSetNPipeSem error: return code = %u\n",rc);
     return 1;
   }

   printf("Waiting for connection to pipe %s...\n",PipeName);

   rc = DosConnectNPipe(PipeHandle);
   if (rc != NO_ERROR) {
     printf("DosConnectNPipe error: return code = %u\n",rc);
     return 1;
   }

   printf("\nCONNECTED\nWaiting for a message...\n");
   rc = DosRead(PipeHandle,            /* Handle of pipe */
                message,               /* Buffer for message read */
                sizeof(message),       /* Buffer size */
                &ulBytes);             /* Number of bytes actually read */
   if (rc != NO_ERROR) {
     printf("DosRead error: return code = %u\n",rc);
     return 1;
   }

   printf("\n\nMessage received was: %s\n\n", message);

   strcpy(message, "Thank you for your message!");
   rc = DosWrite(PipeHandle,           /* Handle of pipe */
                 message,              /* Buffer containing message to write */
                 strlen(message),      /* Length of message */
                 &ulBytes);            /* Number of bytes actually written */
   if (rc != NO_ERROR) {
     printf("DosWrite error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCloseEventSem(hev);
   /* Should check if (rc != NO_ERROR)  here... */

   rc = DosDisConnectNPipe(PipeHandle);
   /* Should check if (rc != NO_ERROR)  here... */

   return NO_ERROR;
}
```

DosCreateNPipe - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosCreatePipe

DosCreatePipe - Syntax

Creates an unnamed pipe.

```
#define INCL_DOSQUEUES
#include <os2.h>

PHFILE    phfRead;   /*  A pointer to an HFILE where the read handle for the pipe is returned. */
PHFILE    phfWrite;  /*  A pointer to an HFILE where the write handle for the pipe is returned. */
ULONG     cb;        /*  The amount of storage to reserve for the pipe. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosCreatePipe(phfRead, phfWrite, cb);
```

DosCreatePipe Parameter - phfRead

phfRead (PHFILE) - output
A pointer to an HFILE where the read handle for the pipe is returned.

DosCreatePipe Parameter - phfWrite

phfWrite (PHFILE) - output
A pointer to an HFILE where the write handle for the pipe is returned.

DosCreatePipe Parameter - cb

cb (ULONG) - input
The amount of storage to reserve for the pipe.

DosCreatePipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreatePipe returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY

For a full list of error codes, see Errors.

DosCreatePipe - Parameters

phfRead (PHFILE) - output
A pointer to an HFILE where the read handle for the pipe is returned.

phfWrite (PHFILE) - output
A pointer to an HFILE where the write handle for the pipe is returned.

cb (ULONG) - input
The amount of storage to reserve for the pipe.

ulrc (APIRET) - returns
Return Code.

DosCreatePipe returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY

For a full list of error codes, see Errors.

DosCreatePipe - Related Functions

Related Functions

- DosClose
- DosDupHandle
- DosRead
- DosWrite

DosCreatePipe - Example Code

This example creates an unnamed pipe, and then closes its read and write handles.

```
#define INCL_DOSQUEUES        /* Queue values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HFILE    ReadHandle  = 0;          /* Read handle of pipe */
HFILE    WriteHandle = 0;          /* Write handle of pipe */
ULONG    PipeSize    = 4096;       /* Size of pipe */
APIRET   rc          = NO_ERROR;   /* API return code */

rc = DosCreatePipe(&ReadHandle, &WriteHandle, PipeSize);

if (rc != NO_ERROR) {
   printf("DosCreatePipe error: return code = %u\n", rc);
   return 1;
}

rc = DosClose(ReadHandle);
if (rc != NO_ERROR) {
   printf("DosClose error: return code = %u\n", rc);
   return 1;
}

rc = DosClose(WriteHandle);
if (rc != NO_ERROR) {
   printf("DosClose error: return code = %u\n", rc);
   return 1;
}

return NO_ERROR;
}
```

DosCreatePipe - Topics

Select an item:

Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosCreateQueue

DosCreateQueue - Syntax

Creates a queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

PHQUEUE    phq;       /*  A pointer to the read/write handle of the queue that is being created. */
```

```
ULONG      priority;  /*  Priority-ordering algorithm flag. */
PSZ        pszName;   /*  A pointer to the ASCIIZ name of the queue. */
APIRET     ulrc;      /*  Return Code. */

ulrc = DosCreateQueue(phq, priority, pszName);
```

DosCreateQueue Parameter - phq

phq (PHQUEUE) - output
A pointer to the read/write handle of the queue that is being created.

After DosCreateQueue returns, this handle may be used immediately by the requesting process; it is not necessary to issue DosOpenQueue.

DosCreateQueue Parameter - priority

priority (ULONG) - input
Priority-ordering algorithm flag.

A set of flags that indicate which priority-ordering algorithm to use when placing elements in the queue, and whether or not to convert to 32-bit addresses the addresses of elements that are placed in the queue by 16-bit processes. Possible values are a combination of the values of the following two flags:

Priority-algorithm flag

QUE_FIFO (0x00000000) FIFO queue.

QUE_LIFO (0x00000001) LIFO queue.

QUE_PRIORITY (0x00000002) Priority queue: the requesting process specifies priority 0 to 15, with 15 being the highest priority.

Address-conversion flag

QUE_NOCONVERT_ADDRESS (0x00000000) The data addresses of elements placed in the queue by 16-bit processes are not converted.

QUE_CONVERT_ADDRESS (0x00000004) The data addresses of elements placed in the queue by 16-bit processes are converted to 32-bit data addresses.

DosCreateQueue Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the queue.

The name string must include \QUEUES\ as the first element of the path. For example, \QUEUES\RETRIEVE\CONTROL.QUE is a valid queue name. This name must be specified by a client process in a DosOpenQueue request before the client process can add an element to the queue.

DosCreateQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateQueue returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
332	ERROR_QUE_DUPLICATE
334	ERROR_QUE_NO_MEMORY
335	ERROR_QUE_INVALID_NAME

For a full list of error codes, see Errors.

DosCreateQueue - Parameters

phq (PHQUEUE) - output
A pointer to the read/write handle of the queue that is being created.

After DosCreateQueue returns, this handle may be used immediately by the requesting process; it is not necessary to issue DosOpenQueue.

priority (ULONG) - input
Priority-ordering algorithm flag.

A set of flags that indicate which priority-ordering algorithm to use when placing elements in the queue, and whether or not to convert to 32-bit addresses the addresses of elements that are placed in the queue by 16-bit processes. Possible values are a combination of the values of the following two flags:

Priority-algorithm flag

QUE_FIFO (0x00000000) FIFO queue.

QUE_LIFO (0x00000001) LIFO queue.

QUE_PRIORITY (0x00000002) Priority queue: the requesting process specifies priority 0 to 15, with 15 being the highest priority.

Address-conversion flag

QUE_NOCONVERT_ADDRESS (0x00000000) The data addresses of elements placed in the queue by 16-bit processes are not converted.

QUE_CONVERT_ADDRESS (0x00000004) The data addresses of elements placed in the queue by 16-bit processes are converted to 32-bit data addresses.

pszName (PSZ) - input
A pointer to the ASCIIZ name of the queue.

The name string must include \QUEUES\ as the first element of the path. For example, \QUEUES\RETRIEVE\CONTROL.QUE is a valid queue name. This name must be specified by a client process in a DosOpenQueue request before the client process can add an element to the queue.

ulrc (APIRET) - returns
Return Code.

DosCreateQueue returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
332	ERROR_QUE_DUPLICATE
334	ERROR_QUE_NO_MEMORY
335	ERROR_QUE_INVALID_NAME

For a full list of error codes, see Errors.

DosCreateQueue - Related Functions

Related Functions

- DosCloseQueue
- DosOpenQueue
- DosPeekQueue
- DosPurgeQueue
- DosQueryQueue
- DosReadQueue
- DosWriteQueue

DosCreateQueue - Example Code

This example creates a FIFO queue named "SPECIAL.QUE". It also opens it, writes to it, reads from it, and finally closes it.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSPROCESS  /* DOS thread and process values */
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

   PSZ         szQueueName  = "\\QUEUES\\SPECIAL.QUE";
   HQUEUE      hqSpecialQue = NULLHANDLE; /* Queue handle                    */
   PSZ         DataBuffer   = "";         /* Data buffer for queue data      */
   REQUESTDATA Request      = {0};        /* Reques */
   PID         pidOwner     = 0;
   ULONG       ulDataLen    = 0;          /* Length of data returned         */
   BYTE        ElemPrty     = 0;          /* Priority of element (returned)  */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateQueue(&hqSpecialQue,    /* Queue handle                    */
            QUE_FIFO |                   /* First-In First-Out order        */
            QUE_CONVERT_ADDRESS,         /* Convert 16-bit addresses to 32  */
            szQueueName);                /* Name of the queue to create     */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosOpenQueue (&pidOwner,         /* PID of queue owner              */
                      &hqSpecialQue,     /* Handle for created queue        */
                      szQueueName);      /* Name of the queue to open       */
   if (rc!= NO_ERROR) {
      printf ("DosOpenQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "To be, or not to be.  That is the question...";
   rc = DosWriteQueue (hqSpecialQue,        /* Queue to write to           */
                       12345L,              /* Request data                */
                       strlen(DataBuffer),  /* Length of data to write     */
                       DataBuffer,          /* Pointer to data             */
                       0L);                 /* Priority (not applicable here) */
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "";                  /* Clear the DataBuffer */
   Request.pid = pidOwner;           /* process ID for the DosReadQueue */

   rc = DosReadQueue (hqSpecialQue,          /* Queue to read from         */
                      &Request,              /* Request data from write    */
                      &ulDataLen,            /* Length of data returned    */
                      (PVOID) &DataBuffer,   /* The data                   */
                      0L,                    /* Remove first element       */
```

```
                    DCWW_WAIT,             /* Wait for available data     */
                    &ElemPrty,             /* Priority of data (returned) */
                    0L);             /* Semaphore to use when not waiting */
   if (rc!= NO_ERROR) {
      printf ("DosReadQueue error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosReadQueue returns: '%s'\n", DataBuffer);
      printf ("      (Request data = %u)\n", Request.ulData);
   }

   rc = DosCloseQueue(hqSpecialQue);         /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosCreateQueue - Topics

Select an item:

Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosCreateThread

DosCreateThread - Syntax

Creates an asynchronous thread of processing under the current process.

```
#define INCL_DOSPROCESS
#include <os2.h>

PTID         ptid;     /*  Address of a doubleword where the thread identifier of the created thread is returned
PFNTHREAD    pfn;      /*  Address of the code to be executed when the thread begins execution. */
ULONG        param;    /*  An argument that is passed to the target thread routine as a parameter. */
ULONG        flag;     /*  Thread flags. */
ULONG        cbStack;  /*  The size, in bytes, of the new thread's stack. */
APIRET       ulrc;     /*  Return Code. */

ulrc = DosCreateThread(ptid, pfn, param, flag,
         cbStack);
```

DosCreateThread Parameter - ptid

ptid (PTID) - output
Address of a doubleword where the thread identifier of the created thread is returned.

DosCreateThread Parameter - pfn

pfn (PFNTHREAD) - input
Address of the code to be executed when the thread begins execution.

This function is called near, accepts a single parameter *param* , and returns a doubleword exit status (see DosExit). Returning from the function without executing DosExit causes the thread to end. In this case, the exit status is the value in the EAX register when the thread ends.

DosCreateThread Parameter - param

param (ULONG) - input
An argument that is passed to the target thread routine as a parameter.

It is usually a pointer to a parameter block.

DosCreateThread Parameter - flag

flag (ULONG) - input
Thread flags.

Possible values are a combination of the following:

CREATE_READY (0x00000000) The new thread starts immediately.

CREATE_SUSPENDED (0x00000001) The thread is created in the suspended state, and the creator of the thread must issue DosResumeThread to start the new thread's execution.

STACK_SPARSE (0x00000000) The system uses the default method for initializing the thread's stack.

STACK_COMMITED (0x00000002) The system precommits all the pages in the stack. One page is 4KB.

DosCreateThread Parameter - cbStack

cbStack (ULONG) - input
The size, in bytes, of the new thread's stack.

The size is rounded up to the nearest page boundary. If you specify a stack size of 0, you still get a one page (4K) stack.

The system allocates the stack upon creation of the thread, and deallocates it upon completion of the thread. The system provides dynamic stack storage commitment up to the limit specified in *cbStack* by using the guard-page technique. See **Remarks** for more details.

DosCreateThread Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosCreateThread returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
95	ERROR_INTERRUPT
115	ERROR_PROTECTION_VIOLATION
164	ERROR_MAX_THRDS_REACHED

For a full list of error codes, see Errors.

DosCreateThread - Parameters

ptid (PTID) - output
Address of a doubleword where the thread identifier of the created thread is returned.

pfn (PFNTHREAD) - input
Address of the code to be executed when the thread begins execution.

This function is called near, accepts a single parameter *param* , and returns a doubleword exit status (see DosExit). Returning from the function without executing DosExit causes the thread to end. In this case, the exit status is the value in the EAX register when the thread ends.

param (ULONG) - input
An argument that is passed to the target thread routine as a parameter.

It is usually a pointer to a parameter block.

flag (ULONG) - input
Thread flags.

Possible values are a combination of the following:

CREATE_READY (0x00000000) The new thread starts immediately.

CREATE_SUSPENDED (0x00000001) The thread is created in the suspended state, and the creator of the thread must issue DosResumeThread to start the new thread's execution.

STACK_SPARSE (0x00000000) The system uses the default method for initializing the thread's stack.

STACK_COMMITED (0x00000002) The system precommits all the pages in the stack. One page is 4KB.

cbStack (ULONG) - input
The size, in bytes, of the new thread's stack.

The size is rounded up to the nearest page boundary. If you specify a stack size of 0, you still get a one page (4K) stack.

The system allocates the stack upon creation of the thread, and deallocates it upon completion of the thread. The system provides dynamic stack storage commitment up to the limit specified in *cbStack* by using the guard-page technique. See **Remarks** for more details.

ulrc (APIRET) - returns
Return Code.

DosCreateThread returns one of the following values:

0	NO_ERROR

8	ERROR_NOT_ENOUGH_MEMORY
95	ERROR_INTERRUPT
115	ERROR_PROTECTION_VIOLATION
164	ERROR_MAX_THRDS_REACHED

For a full list of error codes, see Errors.

DosCreateThread - Remarks

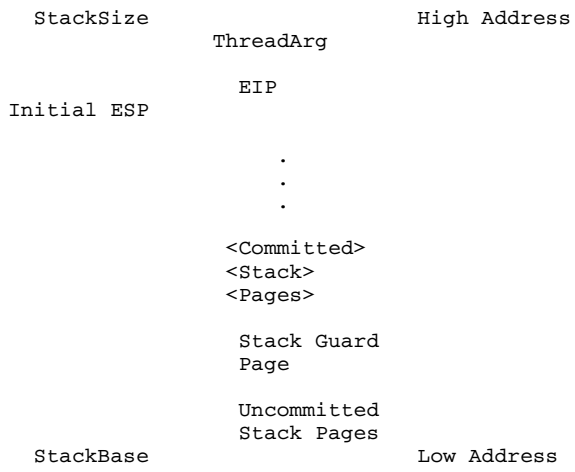
DosCreateThread creates an asynchronous thread of execution under the current process.

The operating system creates the first thread of a process when it starts the executable file. This thread is dispatched with a regular class priority. To start another thread of execution under the current process, the current thread issues DosCreateThread. The thread's initial dispatch point is the address specified for *pfn*. The started thread has a unique stack and register context and the same priority as the requesting thread.

The created thread can access all files and resources owned by the parent process. The thread shares resources with other threads of the process. Any thread in the process can open a file or device, and any other thread can issue a read or write to that handle. This is also true for pipes, queues, and system-managed semaphores.

When a thread is created, the system creates a Thread Information Block (TIB) to maintain per-thread information (TID, priority, and so on) in the user address space. See DosGetInfoBlocks for details on the TIB layout.

When a thread is created, its initial dispatch point is provided by *pfn*. This routine is invoked by Near Call, and when that routine returns or issues DosExit, the thread ends. The format of the thread's stack when the thread begins executing at *pfn* is:



When the system allocates the stack for the thread, a guard page is set up to facilitate dynamic stack growth. When a thread attempts to use stack in or "below" the guard page, a guard-page exception is generated. The default system action for this exception is to attempt to grow the stack by committing another page and moving the guard page. Since only a single guard page is committed at a time, and the page size of the 80386 processor is 4KB, a local stack allocation that is greater than 4KB must be handled by a stack probe that is performed by a compiler-generated routine.

The default stack commitment has one committed page, and a guard page is set up below the committed page. The pages beyond the guard page are uncommitted. If the system cannot allocate another guard page when the guard-page exception is not handled, a guard-page-allocation failure exception is generated. It is essential that applications and language runtime routines handle the guard-page-allocation exception. For more details on guard-page exception management, see DosSetExceptionHandler.

A thread started with DosCreateThread ends upon return of this call or when DosExit is issued. Any thread can temporarily stop the execution of other threads in its process with DosSuspendThread, DosResumeThread, DosEnterCritSec, and DosExitCritSec.

Any thread can also examine and change the priority at which it and other threads execute with DosGetInfoBlocks and DosSetPriority.

DosCreateThread - Related Functions

Related Functions

- DosExit
- DosKillThread
- DosResumeThread
- DosSuspendThread
- DosWaitThread

DosCreateThread - Example Code

The first example creates a new thread within a process, sleeps for 1 second, suspends the thread for 5 seconds, and then waits for the thread to end. The second example passes the address of MyStuff as a pointer to a thread.

Example 1

Compile the examples with MULTITHREAD LIBRARIES. If you are using CSet/2, use the /Gm+ switch.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#define INCL_DOSERRORS      /* DOS error values          */
#include <os2.h>
#include <stdio.h>

void _System CntThreadProc(ULONG LoopMax);  /* Count Thread */

int main(VOID) {
   TID         tidCntThread = 0;     /* ID returned for newly created thread */
   PFNTHREAD   pfnCntThread = &CntThreadProc; /* Address of thread program   */
   ULONG       ulThreadParm = 100;        /* Parameter to thread routine     */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateThread(&tidCntThread,  /* Thread ID (returned by function)  */
                        pfnCntThread,   /* Address of thread program         */
                        ulThreadParm,   /* Parameter passed to ThreadProc    */
                        CREATE_READY |  /* Thread is ready when created      */
                        STACK_SPARSE,   /* Do not pre-commit stack pages     */
                        8192L);         /* Stack size, rounded to page bdy   */
   if (rc != NO_ERROR) {
      printf("DosCreateThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (1000);  /* Sleep for a second to allow thread to run a bit */

   rc = DosSuspendThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosSuspendThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (5000);  /* Sleep 5 seconds before resuming the thread     */

   rc = DosResumeThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosResumeThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosWaitThread (&tidCntThread, DCWW_WAIT);
   if (rc != NO_ERROR) {
      printf ("DosWaitThread error : return code = %u\n", rc);
   }
   printf ("Thread has completed!\n");

   return NO_ERROR;
}

void _System CntThreadProc(ULONG LoopMax )  /* Count thread */
{
   ULONG i = 0;                 /* Loop index */

   for (i=0;i < LoopMax;i++ ) {
      printf ("%d\n", i);
   }
```

```
   return;
}
```

Example 2

This example passes the address of *MyStuff* as a pointer to a thread:

```
  DosCreateThread ( &tid,
                    AFunct,
                    (ULONG) &MyStuff,    /* Cast pointer to ULONG */
                    0L,
                    8192L);
```

When the thread is executed, the ULONG value passed in is a pointer to the structure:

```
  void APIENTRY AFunct (ULONG arg)
    {
    MYSTUFF  *pMyStuff = (MYSTUFF*) arg;
    /* At this point, pMyStuff is the address of the MyStuff structure */
    DosExit (0L,
             0L);
    }
```

DosCreateThread - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosDebug

DosDebug - Syntax

Enables the calling application to control another application for debugging purposes.

```
#define INCL_DOSPROCESS
#include <os2.h>

PVOID     pdbgbuf;  /*  Address of a DosDebug Buffer. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosDebug(pdbgbuf);
```

DosDebug Parameter - pdbgbuf

pdbgbuf (PVOID) - input
Address of a DosDebug Buffer.

DosDebug Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDebug returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
115	ERROR_PROTECTION_VIOLATION
328	ERROR_SYS_INTERNAL (this error code applies to PowerPC only)

For a full list of error codes, see Errors.

DosDebug - Parameters

pdbgbuf (PVOID) - input
Address of a DosDebug Buffer.

ulrc (APIRET) - returns
Return Code.

DosDebug returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
115	ERROR_PROTECTION_VIOLATION
328	ERROR_SYS_INTERNAL (this error code applies to PowerPC only)

For a full list of error codes, see Errors.

DosDebug - Remarks

DosDebug maintains a list of system semaphores which it checks before sending any debug notifications to the debugger. DosDebug allows one process (the debugger) to control the execution of another process that is being debugged (the debuggee).

A process must be selected for debugging when it is started. See DosExecPgm or DosStartSession for how this is done. Once a process has been selected for debugging, you must use DosDebug to control and examine its execution.

If no error is returned, a notification resides in the DosDebug Buffer structure. The *Cmd* field of the DosDebug Buffer structure determines which notification is set. The data returned with the notification varies, depending on the command passed in the *Cmd* field of the DosDebug Buffer structure.

If the return code is set to ERROR_INTERRUPT, a debug notification might have been lost, depending on the command that was interrupted.

DosDebug can also return with a return value set by another function.

DosDebug allows you to view and modify EXE and DLL code and data objects (those objects defined in the EXE header and loaded by the

OS/2 program loader).

In addition to the EXE and DLL code and data objects, DosDebug allows you to view objects allocated dynamically using DosAllocMem. Code or data objects created by other methods may not be accessible.

For details about the commands that are available with DosDebug, see DosDebug Commands.

For details about the notifications that are available with DosDebug, see DosDebug Notifications.

DosDebug - Example Code

The following is NOT a complete C program. It is intended to provide an idea of how to use DosDebug to control another process.

This example illustrates how to modify a word in the process being controlled. It assumes that all the steps have been taken so that the caller controls the other process. It also assumes that the TargetPID, TargetAddr, and NewValue fields have been set appropriately for the operation. */

```
#define INCL_DOSPROCESS   /* Process and thread values */
#define INCL_DOSERRORS    /* Error values */
#include <os2.h>
#include <stdio.h>

uDB_t   DebugBuf   = {0};        /* Debug buffer */
ULONG   TargetPID  = 0;          /* Process ID of controlled process */
ULONG   TargetAddr = 0;          /* Address within the controlled process */
LONG    NewValue   = 0;          /* Value to be substituted */
APIRET  rc         =  NO_ERROR;  /* Return code */

   DebugBuf.Cmd = DBG_C_WriteMem;  /* Perform WRITE WORD command */

   DebugBuf.Pid = TargetPID;       /* Target process to control */

   DebugBuf.Addr = TargetAddr;     /* Target address for command */

   DebugBuf.Value = NewValue;      /* Value to change in other process */

   rc = DosDebug ( &DebugBuf );
   if (rc != NO_ERROR) {
       printf("DosDebug error: return code = %u\n", rc);
       return 1;
   }
```

DosDebug - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Glossary

DosDelete

DosDelete - Syntax

Removes a file name from a directory. The deleted file may be recoverable.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszFile;  /*  Address of the name of the file to be deleted. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosDelete(pszFile);
```

DosDelete Parameter - pszFile

pszFile (PSZ) - input
Address of the name of the file to be deleted.

DosDelete Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDelete returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosDelete - Parameters

pszFile (PSZ) - input
Address of the name of the file to be deleted.

ulrc (APIRET) - returns
Return Code.

DosDelete returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED

87	ERROR_INVALID_PARAMETER
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosDelete - Remarks

Global file-name characters are not permitted in the name of the file to be deleted.

Read-only files cannot be deleted by DosDelete. To delete a read-only file, you must first issue DosSetFileInfo to change the file's read-only attribute to zero, then delete the file.

If a storage directory for the drive has been defined with the SET DELDIR command, the UNDELETE command may recover the deleted file.

DosDelete cannot be used to delete directories. Issue DosDeleteDir to delete a directory.

DosDelete - Related Functions

Related Functions

- DosDeleteDir
- DosForceDelete
- DosSetFileInfo

DosDelete - Example Code

This example creates and deletes a file named "TEST.DAT".

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS error values    */
#include <os2.h>
#include <stdio.h>

int main(void) {
   HFILE  hfFileHandle    = 0L;          /* File Handle */
   ULONG  ulAction        = 0;           /* Action taken */
   UCHAR  uchFileName[20] = "test.dat";  /* File path name */
   APIRET rc              = NO_ERROR;    /* Return code */

   /* Create the file test.dat */

   rc = DosOpen(uchFileName,
                &hfFileHandle,
                &ulAction,
                10L,
                FILE_NORMAL,
                FILE_CREATE,
                OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE,
                0L);
   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf("DosOpen:   File created = %s\n", uchFileName);
   } /* endif */

   rc = DosClose(hfFileHandle);        /* Close the file */
   if (rc != NO_ERROR) {
     printf("DosClose error: return code = %u\n", rc);
     return 1;
   } /* endif */
```

```
   /* Delete file "test.dat" from current directory */

   rc = DosDelete(uchFileName);
   if(rc != NO_ERROR) {
     printf("DosDelete error: return code = %u\n", rc);
     return 1;
   } else {
     printf("DosDelete: File deleted = %s\n", uchFileName);
   } /* endif */

   return NO_ERROR;
}
```

DosDelete - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosDeleteDir

DosDeleteDir - Syntax

Removes a subdirectory from the specified disk.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszDir;  /*  Address of the fully qualified path name of the subdirectory to be removed. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosDeleteDir(pszDir);
```

DosDeleteDir Parameter - pszDir

pszDir (PSZ) - input
Address of the fully qualified path name of the subdirectory to be removed.

DosDeleteDir Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDeleteDir returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
16	ERROR_CURRENT_DIRECTORY
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosDeleteDir - Parameters

pszDir (PSZ) - input
Address of the fully qualified path name of the subdirectory to be removed.

ulrc (APIRET) - returns
Return Code.

DosDeleteDir returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
16	ERROR_CURRENT_DIRECTORY
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosDeleteDir - Remarks

The subdirectory must be empty; that is, it cannot have hidden files or directory entries other than the "." and ".." entries. To delete files, use DosDelete or DosForceDelete.

The root directory and current directory cannot be removed.

DosDeleteDir - Related Functions

Related Functions

- DosDelete
- DosForceDelete

DosDeleteDir - Example Code

This example creates, accesses, and finally deletes a new directory, named "TEMPPROG", from the root directory. Some return codes are not checked for brevity.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   UCHAR   achOrigDirName[256] = "";            /* Original directory name  */
   UCHAR   achNewDirName[256]  = "\\TEMPPROG";  /* New directory to create  */
   UCHAR   achDirName[256]     = "";           /* Directory name for queries */
   ULONG   cbDirPathLen    = 0;                /* Length of directory path   */
   PEAOP2  pEABuf       = NULL;   /* Extended Attribute buffer pointer      */
   ULONG   ulDriveNum   = 0;      /* Drive number: current=0, A=1, B=2, ... */
   APIRET  rc           = NO_ERROR;      /* Return code                      */

   cbDirPathLen = (ULONG) sizeof(achOrigDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achOrigDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Original dir. = \\%s\n", achOrigDirName);

   pEABuf = NULL;  /* Indicate no EAs are to be defined for the directory  */
   rc = DosCreateDir(achNewDirName, pEABuf);   /* Create the new directory */

   if (rc != NO_ERROR) {
      printf("DosCreateDir error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir (achNewDirName);   /* Change to new directory     */

   ulDriveNum = 0;
   cbDirPathLen = (ULONG) sizeof(achDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Current dir.  = \\%s\n", achDirName);

   strcpy(achDirName,"\\");

   rc = DosSetCurrentDir (achDirName);     /* Change to root directory     */
   rc = DosDeleteDir (achNewDirName);      /* Delete the new directory     */

   return NO_ERROR;
}
```

DosDeleteDir - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosDeleteMuxWaitSem

DosDeleteMuxWaitSem - Syntax

Deletes an event semaphore or a mutex semaphore from a muxwait-semaphore list.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMUX      hmux;  /*  The handle of the muxwait semaphore that is to have a semaphore deleted from its semaphore
HSEM      hSem;  /*  The handle of the semaphore that is to be deleted from the semaphore record list of the mux
APIRET    ulrc;  /*  Return Code. */

ulrc = DosDeleteMuxWaitSem(hmux, hSem);
```

DosDeleteMuxWaitSem Parameter - hmux

hmux (HMUX) - input
The handle of the muxwait semaphore that is to have a semaphore deleted from its semaphore record list.

DosDeleteMuxWaitSem Parameter - hSem

hSem (HSEM) - input
The handle of the semaphore that is to be deleted from the semaphore record list of the muxwait semaphore.

DosDeleteMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDeleteMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
286	ERROR_EMPTY_MUXWAIT

For a full list of error codes, see Errors.

DosDeleteMuxWaitSem - Parameters

hmux (HMUX) - input
The handle of the muxwait semaphore that is to have a semaphore deleted from its semaphore record list.

hSem (HSEM) - input
The handle of the semaphore that is to be deleted from the semaphore record list of the muxwait semaphore.

ulrc (APIRET) - returns
Return Code.

DosDeleteMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
286	ERROR_EMPTY_MUXWAIT

For a full list of error codes, see Errors.

DosDeleteMuxWaitSem - Remarks

DosDeleteMuxWaitSem deletes an event semaphore or a mutex semaphore from the existing list of semaphores in a muxwait semaphore.

This function can be called by any thread in the process that created the muxwait semaphore. Other processes can also call this function, but they must first gain access to the muxwait semaphore by calling DosOpenMuxWaitSem.

DosDeleteMuxWaitSem - Related Functions

Related Functions

- DosAddMuxWaitSem
- DosCloseMuxWaitSem
- DosCreateMuxWaitSem
- DosOpenMuxWaitSem
- DosQueryMuxWaitSem
- DosWaitMuxWaitSem

DosDeleteMuxWaitSem - Example Code

This example creates a semaphore record list with five event semaphores. It then creates a MuxWait semaphore with the semaphore record list in its list, waits, deletes the first event semaphore from its list, and closes the MuxWait semaphore.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX      hmuxHandAny = NULLHANDLE;      /* Muxwaithandle */
 HEV       hev[5]      = {0};             /* Event semaphores */
 SEMRECORD apsr[5]     = {{0}};           /* Semaphore records */
 APIRET    rc          = NO_ERROR;        /* Return code */
 ULONG     ulLoop      = 0;               /* Loop count */
 ULONG     ulSem       = 0;

for (ulLoop = 0; ulLoop < 5; ulLoop++) {
    rc = DosCreateEventSem((PSZ) NULL,
                           &hev[ulLoop],
                           0,
```

```
                            FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
    apsr[ulLoop].hsemCur = (HSEM) hev[ulLoop],
    apsr[ulLoop].ulUser = 0;
} /* endfor */

rc = DosCreateMuxWaitSem((PSZ) NULL,
                         &hmuxHandAny,
                         5L,              /* Number of semaphores in list */
                         apsr,            /* Semaphore list */
                         DCMW_WAIT_ANY);  /* Wait for any semaphore */
    if (rc != NO_ERROR) {
      printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosWaitMuxWaitSem(hmuxHandAny,
                       SEM_IMMEDIATE_RETURN,
                       &ulSem);      /* No semaphores have been  posted, so
                                        we should see a timeout below...    */
    if (rc != ERROR_TIMEOUT) {
      printf("DosWaitMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosDeleteMuxWaitSem(hmuxHandAny, apsr[0].hsemCur);
    if (rc != NO_ERROR) {
      printf("DosDeleteMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }
rc = DosCloseMuxWaitSem(hmuxHandAny);
     if (rc != NO_ERROR) {
       printf("DosCloseMuxWaitSem error: return code = %u\n", rc);
       return 1;
     }

return NO_ERROR;
}
```

DosDeleteMuxWaitSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosDevConfig

DosDevConfig - Syntax

Gets information about attached devices.

```
#define INCL_DOSPROCESS
#include <os2.h>

PVOID     pdevinfo;  /*  Address of the area where the information is returned. */
ULONG     item;      /*  Device information to be returned. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosDevConfig(pdevinfo, item);
```

DosDevConfig Parameter - pdevinfo

pdevinfo (PVOID) - output
Address of the area where the information is returned.

All returned device information is BYTE-sized, so this should be the address of a BYTE variable.

DosDevConfig Parameter - item

item (ULONG) - input
Device information to be returned.

This parameter has one of the possible values:

0 DEVINFO_PRINTER
Number of parallel or printer ports.

1 DEVINFO_RS232
Number of RS232 ports.

2 DEVINFO_FLOPPY
Number of diskette drives.

3 DEVINFO_COPROCESSOR
Presence of math coprocessor hardware:

- 0 No coprocessor hardware
- 1 Coprocessor hardware installed

4 DEVINFO_SUBMODEL
PC Submodel Type. The returned information is the system submodel byte.

5 DEVINFO_MODEL
PC Model Type. The returned information is the system model byte.

6 DEVINFO_ADAPTER
Type of primary display adapter:

- 0 Monochrome or printer adapter
- 1 Other

DosDevConfig Return Value - ulrc

ulrc (APIRET) - returns

Return Code.

DosDevConfig returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosDevConfig - Parameters

pdevinfo (PVOID) - output
Address of the area where the information is returned.

All returned device information is BYTE-sized, so this should be the address of a BYTE variable.

item (ULONG) - input
Device information to be returned.

This parameter has one of the possible values:

0 DEVINFO_PRINTER
Number of parallel or printer ports.

1 DEVINFO_RS232
Number of RS232 ports.

2 DEVINFO_FLOPPY
Number of diskette drives.

3 DEVINFO_COPROCESSOR
Presence of math coprocessor hardware:

0	No coprocessor hardware
1	Coprocessor hardware installed

4 DEVINFO_SUBMODEL
PC Submodel Type. The returned information is the system submodel byte.

5 DEVINFO_MODEL
PC Model Type. The returned information is the system model byte.

6 DEVINFO_ADAPTER
Type of primary display adapter:

0	Monochrome or printer adapter
1	Other

ulrc (APIRET) - returns
Return Code.

DosDevConfig returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosDevConfig - Related Functions

Related Functions

- DosBeep

- DosDevIOCtl
- DosPhysicalDisk

DosDevConfig - Example Code

This example gets information about the installed devices and displays it. Some return code checking was omitted for brevity.

```
 #define INCL_DOSDEVICES   /* Device values */
 #define INCL_DOSERRORS    /* Error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

 BYTE    Model      = 0,  /* Model number */
         SubModel   = 0,  /* Submodel */
         Adapter    = 0,  /* Display adapter type */
         CoProc     = 0,  /* Math coprocessor installed? */
         Printers   = 0;  /* Number of printers */

 APIRET  rc         = NO_ERROR; /* Return code */

    rc = DosDevConfig(&Model, DEVINFO_MODEL);
    if (rc != NO_ERROR) {
       printf("DosDevConfig error:  return code = %u\n", rc);
       return 1;
    }

    rc = DosDevConfig(&SubModel, DEVINFO_SUBMODEL);
    printf("    Model/SubModel:  %d / %d\n", Model, SubModel);

    rc = DosDevConfig(&Adapter, DEVINFO_ADAPTER);
    printf("      Display type:  %s\n", (Adapter ? "Color" : "Monochrome"));

    rc = DosDevConfig(&CoProc, DEVINFO_COPROCESSOR);
    printf(" Math co-processor?  %s\n", (CoProc ? "Yes" : "No"));

    rc = DosDevConfig(&Printers, DEVINFO_PRINTER);
    printf("Number of printers:  %d\n", Printers);

    return NO_ERROR;
}
```

DosDevConfig - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosDevIOCtl

DosDevIOCtl - Syntax

Performs control functions on a device specified by an opened device handle.

```
#define INCL_DOSDEVICES
#define INCL_DOSDEVIOCTL
#include <os2.h>

HFILE     hDevice;       /*  Device handle returned by DosOpen, or a standard (open) device handle. */
ULONG     category;      /*  Device category. */
ULONG     function;      /*  Device-specific function code. */
PVOID     pParams;       /*  Address of the command-specific argument list. */
ULONG     cbParmLenMax;  /*  Length, in bytes, of pParams. */
PULONG    pcbParmLen;    /*  Pointer to the length of parameters. */
PVOID     pData;         /*  Address of the data area. */
ULONG     cbDataLenMax;  /*  Length, in bytes, of pData. */
PULONG    pcbDataLen;    /*  Pointer to the length of data. */
APIRET    ulrc;          /*  Return Code. */

ulrc = DosDevIOCtl(hDevice, category, function,
         pParams, cbParmLenMax, pcbParmLen,
         pData, cbDataLenMax, pcbDataLen);
```

DosDevIOCtl Parameter - hDevice

hDevice (HFILE) - input
Device handle returned by DosOpen, or a standard (open) device handle.

DosDevIOCtl Parameter - category

category (ULONG) - input
Device category.

The valid range is 0 to 255.

DosDevIOCtl Parameter - function

function (ULONG) - input
Device-specific function code.

The valid range is 0 to 255.

DosDevIOCtl Parameter - pParams

pParams (PVOID) - input
Address of the command-specific argument list.

DosDevIOCtl Parameter - cbParmLenMax

cbParmLenMax (ULONG) - input
Length, in bytes, of *pParams* .

This is the maximum length of the data to be returned in *pParams* . *pcbParmLen* may be larger than this on input, but not on output.

DosDevIOCtl Parameter - pcbParmLen

pcbParmLen (PULONG) - in/out
Pointer to the length of parameters.

Input	Pointer to the length, in bytes, of the parameters passed in *pParams* . by the application.
Output	Pointer to the length, in bytes, of the parameters returned. If this function returns ERROR_BUFFER_OVERFLOW, then *pcbParmLen* points to the size of the buffer required to hold the parameters returned. No other data is returned in this case.

DosDevIOCtl Parameter - pData

pData (PVOID) - input
Address of the data area.

DosDevIOCtl Parameter - cbDataLenMax

cbDataLenMax (ULONG) - input
Length, in bytes, of *pData* .

This is the maximum length of the data to be returned in *pData* . *pcbDataLen* may be larger than this on input, but not on output.

DosDevIOCtl Parameter - pcbDataLen

pcbDataLen (PULONG) - in/out
Pointer to the length of data.

Input	Pointer to the length, in bytes, of the data passed by the application in *pData* .
Output	Pointer to the length, in bytes, of the data returned. If this function returns ERROR_BUFFER_OVERFLOW, then *pcbDataLen* points to the size of the buffer required to hold the data returned.

DosDevIOCtl Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDevIOCtl returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

For a full list of error codes, see Errors.

DosDevIOCtl - Parameters

hDevice (HFILE) - input
Device handle returned by DosOpen, or a standard (open) device handle.

category (ULONG) - input
Device category.

The valid range is 0 to 255.

function (ULONG) - input
Device-specific function code.

The valid range is 0 to 255.

pParams (PVOID) - input
Address of the command-specific argument list.

cbParmLenMax (ULONG) - input
Length, in bytes, of *pParams* .

This is the maximum length of the data to be returned in *pParams* . *pcbParmLen* may be larger than this on input, but not on output.

pcbParmLen (PULONG) - in/out
Pointer to the length of parameters.

Input	Pointer to the length, in bytes, of the parameters passed in *pParams* . by the application.
Output	Pointer to the length, in bytes, of the parameters returned. If this function returns ERROR_BUFFER_OVERFLOW, then *pcbParmLen* points to the size of the buffer

required to hold the parameters returned. No other data is returned in this case.

pData (PVOID) - input
Address of the data area.

cbDataLenMax (ULONG) - input
Length, in bytes, of *pData* .

This is the maximum length of the data to be returned in *pData* . *pcbDataLen* may be larger than this on input, but not on output.

pcbDataLen (PULONG) - in/out
Pointer to the length of data.

Input	Pointer to the length, in bytes, of the data passed by the application in *pData* .
Output	Pointer to the length, in bytes, of the data returned. If this function returns ERROR_BUFFER_OVERFLOW, then *pcbDataLen* points to the size of the buffer required to hold the data returned.

ulrc (APIRET) - returns
Return Code.

DosDevIOCtl returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

For a full list of error codes, see Errors.

DosDevIOCtl - Remarks

Values returned in the range 0xFF00 through 0xFFFF are user-dependent error codes. Values returned in the range 0xFE00 through 0xFEFF are device-driver-dependent error codes.

This function provides a generic, expandable IOCtl facility.

A null (zero) value for *pData* specifies that this parameter is not defined for the generic IOCtl function being specified. A null value for *pData* causes the values passed in *cbDataLenMax* and *pcbDataLen* to be ignored.

A null (zero) value for *pParams* specifies that this parameter is not defined for the generic IOCtl function being specified. A null value for *pParams* causes the values passed in *cbParmLenMax* and *pcbParmLen* to be ignored.

The kernel formats a generic IOCtl packet and calls the device driver. Because OS/2 Version 1.0 and Version 1.1 device drivers do not understand generic IOCtl packets with *cbDataLenMax* , *pcbDataLen* , *cbParmLenMax* , and *pcbParmLen* , the kernel does not pass these fields to the device driver. Device drivers that are marked as level 2 or higher must support receipt of the generic IOCtl packets with associated length fields.

Do not pass a non-null pointer with a zero length.

See Generic IOCtl Commands for a complete listing of the generic IOCtl control functions (the IOCtl interface).

For debugging considerations, see DosDebug.

DosDevIOCtl - Related Functions

Related Functions

- DosBeep
- DosDevConfig
- DosPhysicalDisk

DosDevIOCtl - Example Code

The following is NOT a complete C program. It is simply intended to provide an idea of how to issue control functions to a device.

This example assumes that DevHandle contains the handle to the device, and that the device recognizes category code hex 83, function code hex 1D and the input parameters and input data area.

```
#define INCL_DOSDEVICES   /* Device values */
#define INCL_DOSERRORS    /* Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

HFILE   DevHandle        = NULLHANDLE;   /* Handle for device */
ULONG   ulCategory       = 0x83;         /* Device category */
ULONG   ulFunction       = 0x1D;         /* Device-specific function */
UCHAR   uchParms[120]    = {0};          /* Input and output for function */
ULONG   ulParmLen        = 0;            /* Input and output parameter size */
UCHAR   uchDataArea[200] = {0};          /* Input and output data area */
ULONG   ulDataLen        = 0;            /* Input and output data size */
APIRET  rc               = NO_ERROR;     /* Return code */

 strcpy(uchParms,"/X /Y /Z");    /* Input parameters */
 ulParmLen = strlen(uchParms);   /* Length of input parameters */

 strcpy(uchDataArea,"DF=123;NP=BCR;UN=1993;MAX=328");  /* Input data */
 ulDataLen = strlen(uchDataArea);                      /* Length of data  */

 rc = DosDevIOCtl(DevHandle,           /* Handle to device */
                  ulCategory,          /* Category of request */
                  ulFunction,          /* Function being requested */
                  uchParms,            /* Input/Output parameter list */
                  sizeof(uchParms),    /* Maximum output parameter size */
                  &ulParmLen,          /* Input:  size of parameter list */
                                       /* Output: size of parameters returned */
                  uchDataArea,         /* Input/Output data area */
                  sizeof(uchDataArea), /* Maximum output data size */
                  &ulDataLen);         /* Input:  size of input data area */
                                       /* Output: size of data returned   */

 if (rc != NO_ERROR) {
     printf("DosDevIOCtl error: return code = %u\n", rc);
     return 1;
 }
```

DosDevIOCtl - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosDisConnectNPipe

DosDisConnectNPipe - Syntax

Acknowledges that a client process has closed a named pipe.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE     hpipe;  /*  The named-pipe handle to disconnect (returned to the server process by DosCreateNPipe). */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosDisConnectNPipe(hpipe);
```

DosDisConnectNPipe Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to disconnect (returned to the server process by DosCreateNPipe).

DosDisConnectNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDisConnectNPipe returns one of the following values:

0	NO_ERROR
109	ERROR_BROKEN_PIPE
230	ERROR_BAD_PIPE

For a full list of error codes, see Errors.

DosDisConnectNPipe - Parameters

hpipe (HPIPE) - input
The named-pipe handle to disconnect (returned to the server process by DosCreateNPipe).

ulrc (APIRET) - returns
Return Code.

DosDisConnectNPipe returns one of the following values:

0	NO_ERROR
109	ERROR_BROKEN_PIPE
230	ERROR_BAD_PIPE

For a full list of error codes, see Errors.

DosDisConnectNPipe - Remarks

DosDisConnectNPipe is issued by a server process to acknowledge that a client process has closed a named pipe. (If a client process tries to issue DosDisConnectNPipe, ERROR_BAD_PIPE is returned.) The pipe cannot be opened by another client process until the server process issues this function, followed by DosConnectNPipe.

Until the client's close has been acknowledged, the server process will receive a value of zero for *pcbActual* (indicating end-of-file) if it tries to read from the pipe, and ERROR_BROKEN_PIPE if it tries to write to it. Clients that attempt to open the pipe receive ERROR_PIPE_BUSY.

Any threads that are blocked on the pipe are awakened by DosDisConnectNPipe. A thread that is blocked on a DosWrite request returns ERROR_BROKEN_PIPE. A thread that is blocked on a DosRead request returns a value of zero for *pcbActual*.

If the client end of the pipe is open when DosDisConnectNPipe is issued, it is forced to close, and the client receives an error code on its next operation. Note that when a client is forced to close in this manner, data may be discarded before it has been read by the client.

DosDisConnectNPipe makes the client's handle invalid, but it does not free the handle. Therefore, a client that is forced off a pipe by DosDisConnectNPipe must still issue DosClose to free the handle resource.

ERROR_BAD_PIPE is returned if you specify an invalid name or file handle.

DosDisConnectNPipe - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosDisConnectNPipe - Example Code

This example handles host end of a named pipe for several other named pipe examples. Some return code checking has been omitted for brevity.

```
#define INCL_BASE
#define INCL_DOSSEMAPHORES
#define INCL_DOSNMPIPES
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
int main(VOID) {
CHAR     PipeName[256] = "\\PIPE\\EXAMPLE" ;     /* Pipe name */
HPIPE    PipeHandle    = NULLHANDLE;             /* Pipe handle */
HEV      hev           = NULLHANDLE;             /* Semaphore handle */
ULONG    ulBytes       = 0;                      /* Bytes read or written */
CHAR     message[256]  = "";                     /* Input/Output buffer */
APIRET   rc            = NO_ERROR;               /* Return code */

   rc = DosCreateNPipe(PipeName,               /* Name of pipe to create */
                       &PipeHandle,            /* Handle returned for pipe */
                       NP_ACCESS_DUPLEX,       /* Duplex pipe */
                       NP_WAIT |
                       NP_TYPE_MESSAGE |
                       NP_READMODE_MESSAGE |
                       NP_WMESG |              /* Write messages */
                       NP_RMESG |              /* Read messages */
                       0x01,                   /* Unique instance of pipe */
                       sizeof(message),        /* Output buffer size */
                       sizeof(message),        /* Input buffer size */
                       0L);                    /* Use default time-out */
   if (rc != NO_ERROR) {
     printf("DosCreateNPipe error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCreateEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev, 0L, 0L);
   /* Should check if (rc != NO_ERROR)  here... */

   rc = DosSetNPipeSem(PipeHandle,       /* Handle for pipe */
                       (HSEM) hev,       /* Handle of semaphore */
                       1L);              /* Used to distinguish among events */
   if (rc != NO_ERROR) {
     printf("DosSetNPipeSem error: return code = %u\n",rc);
     return 1;
   }

   printf("Waiting for connection to pipe %s...\n",PipeName);

   rc = DosConnectNPipe(PipeHandle);
   if (rc != NO_ERROR) {
     printf("DosConnectNPipe error: return code = %u\n",rc);
     return 1;
   }

   printf("\nCONNECTED\nWaiting for a message...\n");
   rc = DosRead(PipeHandle,            /* Handle of pipe */
                message,               /* Buffer for message read */
                sizeof(message),       /* Buffer size */
                &ulBytes);             /* Number of bytes actually read */
   if (rc != NO_ERROR) {
     printf("DosRead error: return code = %u\n",rc);
     return 1;
   }

   printf("\n\nMessage received was: %s\n\n", message);

   strcpy(message, "Thank you for your message!");
   rc = DosWrite(PipeHandle,           /* Handle of pipe */
                 message,              /* Buffer containing message to write */
                 strlen(message),      /* Length of message */
                 &ulBytes);            /* Number of bytes actually written */
   if (rc != NO_ERROR) {
     printf("DosWrite error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCloseEventSem(hev);
   /* Should check if (rc != NO_ERROR)  here... */

   rc = DosDisConnectNPipe(PipeHandle);
   /* Should check if (rc != NO_ERROR)  here... */

   return NO_ERROR;
}
```

DosDisConnectNPipe - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosDupHandle

DosDupHandle - Syntax

Gets a new handle for an open file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;   /*  File handle to duplicate, or alias. */
PHFILE    pHfile;  /*  A pointer to the HFILE in which file handle informtion is stored. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosDupHandle(hFile, pHfile);
```

DosDupHandle Parameter - hFile

hFile (HFILE) - input
File handle to duplicate, or alias.

DosDupHandle Parameter - pHfile

pHfile (PHFILE) - in/out
A pointer to the HFILE in which file handle informtion is stored.

Input	Contents of the address describes how the handle is to be duplicated. Possible values are shown in the list below:	
	0xFFFFFFFF	Allocate a new file handle and return it here.
	Any other value	Assign this value as the new file handle. A valid value is any of the handles assigned to standard I/O, or the handle of a file currently opened by the process.

Output	Contents of the address contains the duplicate file handle. A value of 0xFFFFFFFF returns a value for *pHfile* , allocated by the operating system.

DosDupHandle Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosDupHandle returns one of the following values:

0	NO_ERROR
4	ERROR_TOO_MANY_OPEN_FILES
6	ERROR_INVALID_HANDLE
114	ERROR_INVALID_TARGET_HANDLE

For a full list of error codes, see Errors.

DosDupHandle - Parameters

hFile (HFILE) - input
File handle to duplicate, or alias.

pHfile (PHFILE) - in/out
A pointer to the HFILE in which file handle informtion is stored.

Input	Contents of the address describes how the handle is to be duplicated. Possible values are shown in the list below:	
	0xFFFFFFFF	Allocate a new file handle and return it here.
	Any other value	Assign this value as the new file handle. A valid value is any of the handles assigned to standard I/O, or the handle of a file currently opened by the process.
Output	Contents of the address contains the duplicate file handle. A value of 0xFFFFFFFF returns a value for *pHfile* , allocated by the operating system.	

ulrc (APIRET) - returns
Return Code.

DosDupHandle returns one of the following values:

0	NO_ERROR
4	ERROR_TOO_MANY_OPEN_FILES
6	ERROR_INVALID_HANDLE
114	ERROR_INVALID_TARGET_HANDLE

For a full list of error codes, see Errors.

DosDupHandle - Remarks

Duplicating the handle duplicates and ties all handle-specific information between *hFile* and *pHfile* . For example, if you move the read/write

pointer of either handle with DosRead, DosSetFilePtr, or DosWrite, the pointer for the other handle also is changed.

The valid values for *pHfile* include the following handles for standard I/O, which are always available to the process:

0x00000000	Standard input
0x00000001	Standard output
0x00000002	Standard error.

If a file-handle value of a currently open file is specified in *pHfile* the file handle is closed before it is redefined as the duplicate of *hFile* . Avoid using arbitrary values for *pHfile* .

Issuing DosClose for a file handle does not affect the duplicate handle.

Protected file handles cannot be duplicated.

DosDupHandle - Related Functions

Related Functions

- DosClose
- DosCreatePipe
- DosOpen
- DosRead
- DosSetFHState
- DosSetFilePtr
- DosSetMaxFH
- DosSetRelMaxFH
- DosWrite

DosDupHandle - Example Code

This example creates a pipe and duplicates its read/write handles.

```
#define INCL_DOSQUEUES        /* Queue values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

HFILE    ReadHandle     = NULLHANDLE; /* Read handle of pipe */
HFILE    WriteHandle    = NULLHANDLE; /* Write handle of pipe */
HFILE    NewReadHandle  = (HFILE) -1; /* Duplicate read handle */
HFILE    NewWriteHandle = (HFILE) 10; /* Duplicate write handle */
ULONG    PipeSize       = 42;         /* Size of pipe */
APIRET   rc             = NO_ERROR;   /* API return code */

rc = DosCreatePipe ( &ReadHandle, &WriteHandle, PipeSize );
if (rc != NO_ERROR) {
   printf("DosCreatePipe error: return code = %u\n", rc);
   return 1;
}
      /* Duplicate Read Handle of Pipe - use next available handle */

rc = DosDupHandle ( ReadHandle, &NewReadHandle );
if (rc != NO_ERROR) {
   printf("DosDupHandle error: return code = %u\n", rc);
   return 1;
}
      /* Duplicate Write Handle of Pipe - use handle 10 */

rc = DosDupHandle ( ReadHandle, &NewWriteHandle );
if (rc != NO_ERROR) {
   printf("DosDupHandle error: return code = %u\n", rc);
   return 1;
}
```

```
printf("Handles are: Read:%u  Write:%u  NewRead:%u  NewWrite:%u\n",
            ReadHandle, WriteHandle, NewReadHandle, NewWriteHandle);

return NO_ERROR;
}
```

DosDupHandle - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosEditName

DosEditName - Syntax

Edits file and directory names indirectly by transforming one ASCII string into another, using global file-name characters for editing or search operations on the string.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     metalevel;  /*  The level of editing semantics to use in transforming the source string. */
PSZ       pszSource;  /*  Address of the ASCIIZ string to transform. */
PSZ       pszEdit;    /*  Address of the ASCIIZ string to use for editing. */
PBYTE     pszTarget;  /*  Address of the buffer for the resulting ASCIIZ string. */
ULONG     cbTarget;   /*  The length of the buffer, in bytes, for the resulting string. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosEditName(metalevel, pszSource, pszEdit,
         pszTarget, cbTarget);
```

DosEditName Parameter - metalevel

metalevel (ULONG) - input
The level of editing semantics to use in transforming the source string.

If the value of *metalevel* is 1, the system uses editing semantics for OS/2 Version 1.2. Values greater than 1 are not feasible.

DosEditName Parameter - pszSource

pszSource (PSZ) - input
Address of the ASCIIZ string to transform.

Global file-name characters are specified only in the subdirectory or file-name component of the path name, and are interpreted as search characters. *pszSource* should contain only the component of the path name to edit, not the entire path.

DosEditName Parameter - pszEdit

pszEdit (PSZ) - input
Address of the ASCIIZ string to use for editing.

Global file-name characters specified in the edit string are interpreted as editing characters. Because only the name component of a path name is transformed, this string does not include the path component.

DosEditName Parameter - pszTarget

pszTarget (PBYTE) - output
Address of the buffer for the resulting ASCIIZ string.

DosEditName Parameter - cbTarget

cbTarget (ULONG) - input
The length of the buffer, in bytes, for the resulting string.

DosEditName Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosEditName returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME

For a full list of error codes, see Errors.

DosEditName - Parameters

metalevel (ULONG) - input
The level of editing semantics to use in transforming the source string.

If the value of *metalevel* is 1, the system uses editing semantics for OS/2 Version 1.2. Values greater than 1 are not feasible.

pszSource (PSZ) - input
Address of the ASCIIZ string to transform.

Global file-name characters are specified only in the subdirectory or file-name component of the path name, and are interpreted as search characters. *pszSource* should contain only the component of the path name to edit, not the entire path.

pszEdit (PSZ) - input
Address of the ASCIIZ string to use for editing.

Global file-name characters specified in the edit string are interpreted as editing characters. Because only the name component of a path name is transformed, this string does not include the path component.

pszTarget (PBYTE) - output
Address of the buffer for the resulting ASCIIZ string.

cbTarget (ULONG) - input
The length of the buffer, in bytes, for the resulting string.

ulrc (APIRET) - returns
Return Code.

DosEditName returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME

For a full list of error codes, see Errors.

DosEditName - Remarks

DosEditName is used to search for and edit names of files and subdirectories. Typically, it is used in conjunction with such functions as DosMove and DosCopy, which do not permit the use of global file-name characters, to perform repetitive operations on files.

An example of an editing operation is: *pszSource* = "foo.bar"; *pszEdit* = "*.baz"; result = "FOO.BAZ." In the editing process, the string is changed to uppercase.

Global file-name characters have two uses: searching and editing. If they are specified in *pszSource*, they are interpreted as search characters; in *pszEdit*, they are interpreted as editing characters. This difference can be illustrated with an example using the COPY utility. The user types the following:

```
copy *.old *.new
```

In the source, "*" acts as a search character and determines which files to return to the user. In the target, "*" functions as an editing character by constructing new names for the matched files.

When used as search characters in *pszSource*, global file-name characters simply match files and behave like any other search characters. They have the following meanings:

. The period (.) has no special meaning itself, but "?" gives it one.

* The asterisk will form a match with any character, including a blank, or with the absence of a character. The matching operation does not cross the null character or the backslash (\), which means that only the file name is matched, not an entire path.

? The question mark matches 1 character, unless what it would match is a "." or the terminating null characters, in which case it matches 0 characters. It also does not cross "\".

Any character other than * and ? matches itself, including ".".

Searching is not case-sensitive.

If a file name does not have a period (.), an implicit one is automatically appended to the end during searching operations. For example, searching for "foo." would return "foo".

When used in *pszEdit*, global file-name characters have the following meanings:

. The period (.) in the target synchronizes pointers. It causes the source pointer to match a corresponding pointer to the period in the target. Counting starts from the left of the pointers.

? The question mark copies one character, unless what it would copy is a period (.), in which case it copies no characters. It also copies no characters when the end of the source string is reached.

* The asterisk copies characters from the source to the target until it finds a source character that matches the character following it in the target.

Editing is case-insensitive and case-preserving. If conflicts arise between the case of the source and that of the editing string, the case of the editing string is used, for example:

```
source string:      "file.txt"
editing string:     "*E.TMP"
destination string: "filE.TMP"

copy file.txt  *E.tmp  ->  filE.tmp
```

DosEditName - Related Functions

Related Functions

- DosCopy
- DosMove
- DosQuerySysInfo

DosEditName - Example Code

This example edits the name of the file "CONFIG.SYS", using "*.CPY", and transforms it to "CONFIG.CPY". Then, it copies the contents of the original file to "CONFIG.CPY" using the DosCopy function.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>
 #include <string.h>

int main(VOID) {
   UCHAR   achSourceString[80]  = "config.sys";  /* String to transform */
   UCHAR   achEditString[80]    = "*.cpy";       /* Editing string */
   UCHAR   achTargetString[200] = "";            /* Destination string buffer */
   APIRET  rc                   = NO_ERROR;      /* Return code */

   rc = DosSetDefaultDisk(3);    /* Set drive to C: (1=A, 2=B, 3=C, ...)      */
   if (rc != NO_ERROR) {
      printf("DosSetDefaultDisk error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir ("\\");  /*  Set directory to root */
   if (rc != NO_ERROR) {
      printf("DosSetCurrentDir error: return code = %u\n", rc);
      return 1;
   }

         /* Transform "CONFIG.SYS" using "*.CPY" to "CONFIG.CPY" */

   rc = DosEditName(1, achSourceString, achEditString, achTargetString, 200);
```

```
   if (rc != NO_ERROR) {
      printf("DosEditName error: return code = %u\n", rc);
      return 1;
   }

           /* Copy contents of CONFIG.SYS to the backup file */

   rc = DosCopy(achSourceString,             /* Name of file to be copied    */
                achTargetString,             /* Name of the target file      */
                DCPY_EXISTING);   /* Copy even if target file already exists */

   if (rc != NO_ERROR) {
      printf("DosCopy error: return code = %u\n", rc);
      return 1;
   } else  printf ("Backup file %s created.\n", achTargetString);
   return NO_ERROR;
}
```

DosEditName - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosEnterCritSec

DosEnterCritSec - Syntax

Disables thread switching for the current process.

```
#define INCL_DOSPROCESS
#include <os2.h>

APIRET    ulrc;  /*  Return Code. */

ulrc = DosEnterCritSec();
```

DosEnterCritSec Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosEnterCritSec returns one of the following values:

0	NO_ERROR
309	ERROR_INVALID_THREADID
484	ERROR_CRITSEC_OVERFLOW

For a full list of error codes, see Errors.

DosEnterCritSec - Parameters

ulrc (APIRET) - returns
Return Code.

DosEnterCritSec returns one of the following values:

0	NO_ERROR
309	ERROR_INVALID_THREADID
484	ERROR_CRITSEC_OVERFLOW

For a full list of error codes, see Errors.

DosEnterCritSec - Remarks

DosEnterCritSec causes all other threads in the process to block themselves and give up their time slice. After a DosEnterCritSec request is made, no functions should be called that depend on another thread to do processing until DosExitCritSec has completed. Great care should be taken when using compiler runtime functions and other OS/2 functions after a DosEnterCritSec request has been made, since the underlying processing of the called function may require processing by another thread and thus cause a deadlock.

A thread can also execute code without having to give up time slices to other threads in its process if it requests a priority class that is higher than those of the other threads. A thread's priority is examined with DosGetInfoBlocks, and changed with DosSetPriority.

A count is maintained of the number of times DosEnterCritSec is issued without a corresponding DosExitCritSec. The count is incremented by DosEnterCritSec and decremented by DosExitCritSec. Normal thread dispatching is not restored until the count is zero. The outstanding DosEnterCritSec count is maintained in a word. If an overflow occurs, the count is set to the maximum value, no operation is performed, and the request returns with ERROR_CRITSEC_OVERFLOW.

If a signal occurs, thread 1 begins execution to process the signal even though another thread in the process has a DosEnterCritSec active. Thread 1 of a process is its initial thread of execution, not a thread created with DosCreateThread. Any processing done by thread 1 to satisfy the signal must not include accessing the critical resource intended to be protected by DosEnterCritSec.

Note: This function is very powerful and must be used with caution! It should be used only in a most cooperative environment where the state of all threads in known. While in the critical section, do not call other compiler runtime or OS/2 functions that could start another thread that it would depend on running before being able to return.

DosQueryThreadContext can be used to obtain the context of other threads in the process once they have been blocked by DosEnterCritSec.

ERROR_INVALID_THREADID is returned when an invalid attempt is made to enter a critical section of code in a signal handler or exception handler.

ERROR_INVALID_THREADID is also returned when a dynamic link library (DLL) routine incorrectly issues DosEnterCritSec.

DosEnterCritSec - Related Functions

Related Functions

- DosCreateThread
- DosExitCritSec

- DosQueryThreadContext

DosEnterCritSec - Example Code

This example shows how a thread enters and exits a critical section of code.

```
#define INCL_DOSPROCESS      /* Process values */
#define INCL_DOSERRORS       /* Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID)
  {
  APIRET  rc = NO_ERROR;    /* Return code */

   rc = DosEnterCritSec();

   if (rc != NO_ERROR) {
       printf("DosEnterCritSec error: return code = %u\n",rc);
       return 1;
   }
     /*********************************************/
     /* Add critical section code here.  While this */
     /* code is running, all other threads are      */
     /* stopped.  CALL NO LIBRARY OR OS/2 FUNCTIONS */
     /* HERE UNLESS YOU KNOW THEY DO NOT REQUIRE    */
     /* ACTION BY ANOTHER THREAD IN THE PROCESS.    */
     /*********************************************/

   rc =  DosExitCritSec();

   if (rc != NO_ERROR) {
       printf("DosExitCritSec error: return code = %u\n",rc);
       return 1;
   }

   return NO_ERROR;
   }
```

DosEnterCritSec - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosEnterMustComplete

DosEnterMustComplete - Syntax

Provides entry into a section of code in which asynchronous exceptions are held.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PULONG    pulNesting;  /*  A pointer to a value that is equal to the number of DosEnterMustComplete requests min
APIRET    ulrc;        /*  Return Code. */

ulrc = DosEnterMustComplete(pulNesting);
```

DosEnterMustComplete Parameter - pulNesting

pulNesting (PULONG) - output
A pointer to a value that is equal to the number of DosEnterMustComplete requests minus the number of DosExitMustComplete requests for the current thread.

DosEnterMustComplete Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosEnterMustComplete returns one of the following values:

0	NO_ERROR
650	ERROR_NESTING_TOO_DEEP

For a full list of error codes, see Errors.

DosEnterMustComplete - Parameters

pulNesting (PULONG) - output
A pointer to a value that is equal to the number of DosEnterMustComplete requests minus the number of DosExitMustComplete requests for the current thread.

ulrc (APIRET) - returns
Return Code.

DosEnterMustComplete returns one of the following values:

0	NO_ERROR
650	ERROR_NESTING_TOO_DEEP

For a full list of error codes, see Errors.

DosEnterMustComplete - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosEnterMustComplete notifies the system that the thread is entering a section of code in which asynchronous exceptions (signals and asynchronous process terminations) are to be held, rather than being immediately delivered to the thread.

For a detailed list of the system exceptions, see System Exceptions.

DosEnterMustComplete - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosExitMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler
- DosUnwindException

DosEnterMustComplete - Example Code

This example shows how a thread can notify the system to hold asynchronous exceptions during a section of code.

```
#define INCL_DOSEXCEPTIONS    /* Exception values */
#define INCL_DOSERRORS        /* Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID)
  {
  ULONG   ulNestLevel = 0;  /* Global variable tracking nesting
                               of DosEnterMustComplete calls     */
  APIRET  rc = NO_ERROR;    /* Return code                       */

   rc = DosEnterMustComplete(&ulNestLevel);

   if (rc != NO_ERROR) {
       printf("DosEnterMustComplete error: return code = %u\n",rc);
       return 1;
   } else {
       printf("ulNestLevel = %u\n",ulNestLevel);
   }

       /* ADD BLOCK OF CODE THAT MUST COMPLETE HERE... */

   rc = DosExitMustComplete(&ulNestLevel);

   if (rc != NO_ERROR) {
       printf("DosExitMustComplete error: return code = %u\n",rc);
       return 1;
   } else {
       printf("ulNestLevel = %u\n",ulNestLevel);
   }

   return NO_ERROR;
   }
```

DosEnterMustComplete - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosEnumAttribute

DosEnumAttribute - Syntax

Identifies names and lengths of extended attributes for a specific file or subdirectory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     ulRefType;    /*  A value that indicates whether pvFile points to a handle or to an ASCIIZ name. */
PVOID     pvFile;       /*  Address of the handle of a file returned by DosOpen; or the ASCIIZ name of a file or
ULONG     ulEntry;      /*  Ordinal of an entry in the file object's EA list, which indicates where in the list
PVOID     pvBuf;        /*  Address of the buffer where EA information is returned. */
ULONG     cbBuf;        /*  The length, in bytes, of the buffer pointed to by pvBuf. */
PULONG    pulCount;     /*  Pointer to number of EAs. */
ULONG     ulInfoLevel;  /*  Level of information required. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosEnumAttribute(ulRefType, pvFile,
         ulEntry, pvBuf, cbBuf, pulCount,
         ulInfoLevel);
```

DosEnumAttribute Parameter - ulRefType

ulRefType (ULONG) - input
A value that indicates whether *pvFile* points to a handle or to an ASCIIZ name.

Possible values are shown in the following list:

0	ENUMEA_REFTYPE_FHANDLE Handle of a file.
1	ENUMEA_REFTYPE_PATH ASCIIZ name of a file or subdirectory.

DosEnumAttribute Parameter - pvFile

pvFile (PVOID) - input
Address of the handle of a file returned by DosOpen; or the ASCIIZ name of a file or subdirectory.

DosEnumAttribute Parameter - ulEntry

ulEntry (ULONG) - input
Ordinal of an entry in the file object's EA list, which indicates where in the list to begin the return of EA information.

The value 0 is reserved. A value of 1 indicates the file object's first EA; a value of 2, the second; and so on.

DosEnumAttribute Parameter - pvBuf

pvBuf (PVOID) - output
Address of the buffer where EA information is returned.

Level 1 information (*ulInfoLevel* == ENUMEA_LEVEL_NO_VALUE) is returned in a data structure of type DENA2.

DosEnumAttribute Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of the buffer pointed to by *pvBuf* .

DosEnumAttribute Parameter - pulCount

pulCount (PULONG) - in/out
Pointer to number of EAs.

Input	A pointer to the ULONG containing the number of EAs for which information is requested. A value of -1 requests that information be returned for as many EAs whose information fits in *pvBuf* .
Output	A pointer to the ULONG containing the actual number of EAs for which information is returned. When this value is greater than 1, enumerated information is returned in a series of DENA2 data structures. Each data structure is aligned on a doubleword (or 32-bit) boundary. The first field of the data structure *(oNextEntryOffset*) contains the number of bytes to the beginning of the next data structure. A value of zero for *oNextEntryOffset* indicates that this is the last data structure.

DosEnumAttribute Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input

Level of information required.

Only the value 1 (ENUMEA_LEVEL_NO_VALUE) can be specified, indicating return of level 1 information.

DosEnumAttribute Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosEnumAttribute returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosEnumAttribute - Parameters

ulRefType (ULONG) - input
A value that indicates whether *pvFile* points to a handle or to an ASCIIZ name.

Possible values are shown in the following list:

0	ENUMEA_REFTYPE_FHANDLE Handle of a file.
1	ENUMEA_REFTYPE_PATH ASCIIZ name of a file or subdirectory.

pvFile (PVOID) - input
Address of the handle of a file returned by DosOpen; or the ASCIIZ name of a file or subdirectory.

ulEntry (ULONG) - input
Ordinal of an entry in the file object's EA list, which indicates where in the list to begin the return of EA information.

The value 0 is reserved. A value of 1 indicates the file object's first EA; a value of 2, the second; and so on.

pvBuf (PVOID) - output
Address of the buffer where EA information is returned.

Level 1 information (*ulInfoLevel* == ENUMEA_LEVEL_NO_VALUE) is returned in a data structure of type DENA2.

cbBuf (ULONG) - input
The length, in bytes, of the buffer pointed to by *pvBuf*.

pulCount (PULONG) - in/out
Pointer to number of EAs.

Input	A pointer to the ULONG containing the number of EAs for which information is requested. A value of -1 requests that information be returned for as many EAs whose information fits in *pvBuf*.
Output	A pointer to the ULONG containing the actual number of EAs for which information is returned. When this value is greater than 1, enumerated information is returned in a series of DENA2 data structures. Each

data structure is aligned on a doubleword (or 32-bit) boundary. The first field of the data structure *(oNextEntryOffset*) contains the number of bytes to the beginning of the next data structure. A value of zero for *oNextEntryOffset* indicates that this is the last data structure.

ulInfoLevel (ULONG) - input
Level of information required.

Only the value 1 (ENUMEA_LEVEL_NO_VALUE) can be specified, indicating return of level 1 information.

ulrc (APIRET) - returns
Return Code.

DosEnumAttribute returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosEnumAttribute - Remarks

The structure that DosEnumAttribute returns is used to calculate the size of the buffer needed to hold the full extended attribute (FEA2) information for a DosQueryPathInfo or DosQueryFileInfo call that actually gets the FEA2. The buffer size is calculated as follows:

Four bytes (for *oNextEntryOffset*) +
One byte (for *fEA*) +
One byte (for *cbName*) +
Two bytes (for *cbValue*) +
Value of *cbName* (for the name of the EA) +
One byte (for terminating NULL in *cbName*) +
Value of *cbValue* (for the value of the EA)

Each entry must start on a doubleword boundary.

A process can continue through a file's EA list by reissuing DosEnumAttribute with *ulEntry* set to the value specified in the previous call, plus the value returned in *pulCount* .

DosEnumAttribute does not control the specific ordering of EAs; it merely identifies them. Extended attributes can have multiple readers and writers, just as the files they are associated with can. If a file is open in a sharing mode that allows other processes to modify the file's EA list, repetitively calling DosEnumAttribute to back up to an EA's position may return inconsistent results. For example with DosSetFileInfo or DosSetPathInfo, another process can edit the EA list between calls by your process to DosEnumAttribute. Therefore, the EA returned when *ulEntry* is 11 for the first call might not be the same EA returned when *ulEntry* is 11 for the next call.

To prevent EAs from being modified between calls to DosEnumAttribute for a specified file handle or file name, the calling function must open the file in deny-write sharing mode before it calls DosEnumAttribute. If a subdirectory name is specified, modification by other processes is not a concern, because no sharing is possible.

When a value of 1 (ENUMEA_REFTYPE_PATH) is specified for *ulRefType* , the EAs returned are current only when the call was made, and may have been changed by another thread or process since then.

DosEnumAttribute - Related Functions

Related Functions

- DosCreateDir
- DosOpen
- DosQueryFileInfo

- DosQueryPathInfo
- DosSetFileInfo
- DosSetPathInfo

DosEnumAttribute - Example Code

This example gets all the possible extended attributes (EAs) from the file "ATTRIB.EXE" in the OS2 directory, and then displays the names.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {
   UCHAR    EnumBuf[200] = {0};      /* Data Buffer */
   ULONG    ulEnumCnt    = 0;        /* Count of entries to return */
   FEA2     *ptr         = NULL;     /* Pointer to data items returned */
   ULONG    ulTemp       = 0;
   APIRET   rc           = NO_ERROR; /* Return code  */
   ULONG    i            = 0;        /* Loop index */

   ulEnumCnt = (ULONG)-1; /* Request as many attributes as will fit in buffer */

   rc = DosEnumAttribute(ENUMEA_REFTYPE_PATH,   /* ASCIIZ string to be used   */
                         "c:\\os2\\attrib.exe", /* Name of file               */
                         1L,                    /* Start with first attribute */
                         (PVOID)&EnumBuf,       /* Buffer for information     */
                         sizeof(EnumBuf),
                         &ulEnumCnt,
                         ENUMEA_LEVEL_NO_VALUE); /* Request level 1 info      */

   if (rc != NO_ERROR) {
      printf("DosEnumAttribute error: return code = %u\n", rc);
      return 1;
   }

   ptr = (FEA2 *)EnumBuf; /* Mask the buffer pointer to an FEA2 structure */

   printf ("Attributes found = %u\n", ulEnumCnt);

   for (i = 0; i < ulEnumCnt; i++) {
      printf ("name = %s\n", ptr->szName);         /* increment the ptr     */
      ulTemp = ptr->oNextEntryOffset + (ULONG)ptr; /* with the value in     */
      ptr = (FEA2 *)ulTemp;                        /* oNextEntryOffset      */
   } /* endfor */                                  /* to access next record */

   return NO_ERROR;
}
```

DosEnumAttribute - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosErrClass

DosErrClass - Syntax

Provides more information about return values that have been received from other control-program functions.

```
#define INCL_DOSMISC
#include <os2.h>

ULONG     code;     /*  A nonzero return value returned by a control-program function. */
PULONG    pClass;   /*  A pointer to the ULONG in which the classification for the error is returned. */
PULONG    pAction;  /*  A pointer to the ULONG in which the recommended corrective action for the error is retur
PULONG    pLocus;   /*  A pointer to the ULONG in which the error origination is returned. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosErrClass(code, pClass, pAction,
         pLocus);
```

DosErrClass Parameter - code

code (ULONG) - input
A nonzero return value returned by a control-program function.

A nonzero return value indicates that an error has occurred.

DosErrClass Parameter - pClass

pClass (PULONG) - output
A pointer to the ULONG in which the classification for the error is returned.

The following values are possible:

1	ERRCLASS_OUTRES Out of resources
2	ERRCLASS_TEMPSIT Temporary situation
3	ERRCLASS_AUTH Authorization failed
4	ERRCLASS_INTRN Internal error
5	ERRCLASS_HRDFAIL Device hardware failure
6	ERRCLASS_SYSFAIL System failure
7	ERRCLASS_APPERR Probable application error

8	ERRCLASS_NOTFND Item not located
9	ERRCLASS_BADFMT Bad format for function or data
10	ERRCLASS_LOCKED Resource or data locked
11	ERRCLASS_MEDIA Incorrect media, cyclic redundancy check (CRC) error
12	ERRCLASS_ALREADY Action already taken or done, or resource already exists
13	ERRCLASS_UNK Unclassified
14	ERRCLASS_CANT Cannot perform requested action
15	ERRCLASS_TIME Timeout

DosErrClass Parameter - pAction

pAction (PULONG) - output
A pointer to the ULONG in which the recommended corrective action for the error is returned.

The following values are possible:

1	ERRACT_RETRY Retry immediately
2	ERRACT_DLYRET Delay and retry
3	ERRACT_USER Bad user input - get new values
4	ERRACT_ABORT Terminate in an orderly manner
5	ERRACT_PANIC Terminate immediately
6	ERRACT_IGNORE Ignore error
7	ERRACT_INTRET Retry after user intervention

DosErrClass Parameter - pLocus

pLocus (PULONG) - output
A pointer to the ULONG in which the error origination is returned.

The following values are possible:

1	ERRLOC_UNK Unknown

2 ERRLOC_DISK
Random-access device such as a disk

3 ERRLOC_NET
Network

4 ERRLOC_SERDEV
Serial device

5 ERRLOC_MEM
Memory

DosErrClass Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosErrClass returns one of the following value:

0 NO_ERROR

For a full list of error codes, see Errors.

DosErrClass - Parameters

code (ULONG) - input
A nonzero return value returned by a control-program function.

A nonzero return value indicates that an error has occurred.

pClass (PULONG) - output
A pointer to the ULONG in which the classification for the error is returned.

The following values are possible:

1 ERRCLASS_OUTRES
Out of resources

2 ERRCLASS_TEMPSIT
Temporary situation

3 ERRCLASS_AUTH
Authorization failed

4 ERRCLASS_INTRN
Internal error

5 ERRCLASS_HRDFAIL
Device hardware failure

6 ERRCLASS_SYSFAIL
System failure

7 ERRCLASS_APPERR
Probable application error

8 ERRCLASS_NOTFND
Item not located

9 ERRCLASS_BADFMT
Bad format for function or data

10	ERRCLASS_LOCKED Resource or data locked
11	ERRCLASS_MEDIA Incorrect media, cyclic redundancy check (CRC) error
12	ERRCLASS_ALREADY Action already taken or done, or resource already exists
13	ERRCLASS_UNK Unclassified
14	ERRCLASS_CANT Cannot perform requested action
15	ERRCLASS_TIME Timeout

pAction (PULONG) - output
A pointer to the ULONG in which the recommended corrective action for the error is returned.

The following values are possible:

1	ERRACT_RETRY Retry immediately
2	ERRACT_DLYRET Delay and retry
3	ERRACT_USER Bad user input - get new values
4	ERRACT_ABORT Terminate in an orderly manner
5	ERRACT_PANIC Terminate immediately
6	ERRACT_IGNORE Ignore error
7	ERRACT_INTRET Retry after user intervention

pLocus (PULONG) - output
A pointer to the ULONG in which the error origination is returned.

The following values are possible:

1	ERRLOC_UNK Unknown
2	ERRLOC_DISK Random-access device such as a disk
3	ERRLOC_NET Network
4	ERRLOC_SERDEV Serial device
5	ERRLOC_MEM Memory

ulrc (APIRET) - returns
Return Code.

DosErrClass returns one of the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosErrClass - Remarks

DosErrClass receives a nonzero return value from another control-program function as input. It then classifies the return value, tells where in the system the error occurred, and recommends a corrective action.

When called by a family-mode application, DosErrClass can return a valid error classification only for errors that have actually occurred. Also, the classifications of a given return value might not be the same for family-mode and OS/2-mode applications.

DosErrClass - Related Functions

Related Functions

- DosError

DosErrClass - Example Code

This example shows how DosError and DosErrClass handle errors. Run this program without a diskette in drive A: so that the DosOpen call fails with a "device not ready" condition, or run it with a disk there for a "file not found" condition.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#define INCL_DOSMISC      /* DOS miscellaneous   */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR        uchFileName[80] = "A:\\NONEXIST.ZQX"; /* File to fail on      */
HFILE        hfConfig      = 0;             /* Handle for Config file      */
ULONG        ulOpenAction  = 0,             /* Action taken by DosOpen     */
             ulErrorClass  = 0,             /* Type of error encountered   */
             ulErrorAction = 0,             /* Action warranted by error   */
             ulErrorLoc    = 0;             /* Where error occurred        */
APIRET       rc            = NO_ERROR,      /* Return code                 */
             DosOpenRc     = NO_ERROR;      /* Return code from DosOpen    */

 rc = DosError(FERR_DISABLEHARDERR); /* Suppress error window.  If this is
                                        omitted, OS/2 will put up an error
                                        window asking the user to intervene */
 if (rc != NO_ERROR) {
    printf("DosError error: return code = %u\n", rc);
    return 1;
 }
 DosOpenRc = DosOpen(uchFileName,          /* File to open (path and name) */
              &hfConfig,                   /* File handle returned         */
              &ulOpenAction,               /* Action taken by DosOpen      */
              0L,0L,        /* Primary allocation and attributes (ignored) */
              OPEN_ACTION_FAIL_IF_NEW |
              OPEN_ACTION_OPEN_IF_EXISTS,  /* Open an existing file only   */
              OPEN_SHARE_DENYNONE   |
              OPEN_ACCESS_READWRITE,       /* Read and write access        */
              0L);                         /* Extended attributes(ignored) */

   if (DosOpenRc == NO_ERROR) {
       printf("DosOpen successful... but it should have failed.\n");
       return 1;
   } else {
     rc = DosErrClass(DosOpenRc,           /* Return code from failing API */
                      &ulErrorClass,       /* Class of error               */
                      &ulErrorAction,      /* Action to take for error     */
                      &ulErrorLoc);        /* Where did the error occur?   */

     if (rc != NO_ERROR) {
         printf("DosErrClass error: return code = %u\n", rc);
         return 1;
     } else {
```

```
          printf("rc= %u  ErrorClass= %u  ErrorLoc= %u\n",
                  DosOpenRc, ulErrorClass, ulErrorLoc);

          printf("ErrorAction= %u means: ", ulErrorAction);
          switch (ulErrorAction) {
           case(ERRACT_RETRY):  printf("Retry immediately");    break;
           case(ERRACT_DLYRET): printf("Retry after a delay");  break;
           case(ERRACT_USER):   printf("Incorrect user input"); break;
           case(ERRACT_ABORT):  printf("Terminate in an orderly fashion");
                                                               break;
           case(ERRACT_PANIC):  printf("Terminate NOW");        break;
           case(ERRACT_IGNORE): printf("Ignore this error");    break;
           case(ERRACT_INTRET): printf("Retry after user intervenes");
                                                               break;
           default:             printf("Unknown error action"); break;
          } /* switch */
          printf("\n");
       }
    }  /* DosOpen failure */

  rc = DosError(FERR_ENABLEHARDERR);        /* Re-enable error window */
  if (rc != NO_ERROR) {
     printf("DosError error: return code = %u\n", rc);
     return 1;
  }

  return NO_ERROR;
}
```

DosErrClass - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosError

DosError - Syntax

Disables or enables error notification to end users for errors in the current process.

```
#define INCL_DOSMISC
#include <os2.h>

ULONG     error;  /*  Error and Exception pop-up flags. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosError(error);
```

DosError Parameter - error

error (ULONG) - input
Error and Exception pop-up flags.

The unused high-order bits are reserved, and must be zero. The following values can be specified for this parameter. They can be combined using the "logical or" (|) operator.

FERR_DISABLEHARDERR (0x00000000)
Disable hard error pop-ups.

FERR_ENABLEHARDERR (0x00000001)
Enable hard error pop-ups.

FERR_ENABLEEXCEPTION (0x00000000)
Enable program exception and untrapped numeric-processor exception pop-ups.

FERR_DISABLEEXCEPTION (0x00000002)
Disable program exception and untrapped numeric-processor exception pop-ups.

DosError Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosError returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosError - Parameters

error (ULONG) - input
Error and Exception pop-up flags.

The unused high-order bits are reserved, and must be zero. The following values can be specified for this parameter. They can be combined using the "logical or" (|) operator.

FERR_DISABLEHARDERR (0x00000000)
Disable hard error pop-ups.

FERR_ENABLEHARDERR (0x00000001)
Enable hard error pop-ups.

FERR_ENABLEEXCEPTION (0x00000000)
Enable program exception and untrapped numeric-processor exception pop-ups.

FERR_DISABLEEXCEPTION (0x00000002)
Disable program exception and untrapped numeric-processor exception pop-ups.

ulrc (APIRET) - returns
Return Code.

DosError returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosError - Remarks

DosError disables or enables end-user notification of hard errors, program exceptions, or untrapped, numeric-processor exceptions for the current process. This affects the notifications for all threads in the current process.

The CONFIG.SYS statements AUTOFAIL=ON and SUPPRESSPOPUPS=x can be used to disable these notifications for all processes in the system.

If DosError is not issued, and AUTOFAIL=ON and SUPPRESSPOPUPS=x have not been specified in CONFIG.SYS, user notification for hard errors and exceptions is enabled.

There is no corresponding function to query the current status of error notification. Care should be taken if this function is used in a DLL, since the caller may have already disabled end-user error notifications and the DLL could inadvertently enable them.

DosError - Related Functions

Related Functions

- DosErrClass

DosError - Example Code

This example shows how DosError and DosErrClass handle errors. Run this program without a diskette in drive A: so that the DosOpen function call fails with a "device not ready" condition, or run it with a disk there for a "file not found" condition.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#define INCL_DOSMISC      /* DOS miscellaneous   */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR        uchFileName[80] = "A:\\NONEXIST.ZQX"; /* File to fail on      */
HFILE        hfConfig      = 0;             /* Handle for Config file      */
ULONG        ulOpenAction  = 0,             /* Action taken by DosOpen     */
             ulErrorClass  = 0,             /* Type of error encountered   */
             ulErrorAction = 0,             /* Action warranted by error   */
             ulErrorLoc    = 0;             /* Where error occurred        */
APIRET       rc            = NO_ERROR,      /* Return code                 */
             DosOpenRc     = NO_ERROR;      /* Return code from DosOpen    */

 rc = DosError(FERR_DISABLEHARDERR); /* Suppress error window.  If this is
                                        omitted, OS/2 will put up an error
                                        window asking the user to intervene */
 if (rc != NO_ERROR) {
    printf("DosError error: return code = %u\n", rc);
    return 1;
 }

 DosOpenRc = DosOpen(uchFileName,          /* File to open (path and name) */
              &hfConfig,                   /* File handle returned         */
              &ulOpenAction,               /* Action taken by DosOpen      */
              0L,0L,        /* Primary allocation and attributes (ignored) */
              OPEN_ACTION_FAIL_IF_NEW |
              OPEN_ACTION_OPEN_IF_EXISTS,  /* Open an existing file only   */
              OPEN_SHARE_DENYNONE   |
```

```
              OPEN_ACCESS_READWRITE,       /* Read and write access        */
              0L);                         /* Extended attributes(ignored) */

    if (DosOpenRc == NO_ERROR) {
        printf("DosOpen successful... but it should have failed.\n");
        return 1;
    } else {
      rc = DosErrClass(DosOpenRc,          /* Return code from failing API */
                       &ulErrorClass,      /* Class of error               */
                       &ulErrorAction,     /* Action to take for error     */
                       &ulErrorLoc);       /* Where did the error occur?   */

      if (rc != NO_ERROR) {
          printf("DosErrClass error: return code = %u\n", rc);
          return 1;
      } else {
          printf("rc= %u  ErrorClass= %u  ErrorLoc= %u\n",
                  DosOpenRc, ulErrorClass, ulErrorLoc);
          printf("ErrorAction= %u means: ", ulErrorAction);
          switch (ulErrorAction) {
           case(ERRACT_RETRY):  printf("Retry immediately");    break;
           case(ERRACT_DLYRET): printf("Retry after a delay");  break;
           case(ERRACT_USER):   printf("Incorrect user input"); break;
           case(ERRACT_ABORT):  printf("Terminate in an orderly fashion");
                                                                break;
           case(ERRACT_PANIC):  printf("Terminate NOW");        break;
           case(ERRACT_IGNORE): printf("Ignore this error");    break;
           case(ERRACT_INTRET): printf("Retry after user intervenes");
                                                                break;
           default:             printf("Unknown error action"); break;
          } /* switch */
          printf("\n");
      }
   }  /* DosOpen failure */

  rc = DosError(FERR_ENABLEHARDERR);       /* Re-enable error window */
  if (rc != NO_ERROR) {
     printf("DosError error: return code = %u\n", rc);
     return 1;
  }

  return NO_ERROR;
}
```

DosError - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosExecPgm

DosExecPgm - Syntax

Lets a program run another program as a child process.

```
#define INCL_DOSPROCESS
#include <os2.h>

PCHAR           pObjname;   /*  Address of the buffer in which the name of the object that contributed to the fa
LONG            cbObjname;  /*  Length, in bytes, of the buffer described by pObjname. */
ULONG           execFlag;   /*  Flag indicating how the program runs in relation to the requester, and whether e
PSZ             pArg;       /*  Address of the ASCIIZ argument strings passed to the program. */
PSZ             pEnv;       /*  Address of the ASCIIZ environment strings passed to the program. */
PRESULTCODES    pRes;       /*  Pointer to the RESULTCODES structure where the process ID, or the termination co
PSZ             pName;      /*  Address of the name of the file that contains the program to be executed. */
APIRET          ulrc;       /*  Return Code. */

ulrc = DosExecPgm(pObjname, cbObjname, execFlag,
         pArg, pEnv, pRes, pName);
```

DosExecPgm Parameter - pObjname

pObjname (PCHAR) - output
Address of the buffer in which the name of the object that contributed to the failure of DosExecPgm is returned.

DosExecPgm Parameter - cbObjname

cbObjname (LONG) - input
Length, in bytes, of the buffer described by *pObjname* .

DosExecPgm Parameter - execFlag

execFlag (ULONG) - input
Flag indicating how the program runs in relation to the requester, and whether execution is under conditions for debugging.

The values of this field are shown in the following list:

0 EXEC_SYNC
Execution is synchronous to the parent process. The termination code and result code are stored in the RESULTCODES structure pointed to by *pRes* .

1 EXEC_ASYNC
Execution is asynchronous to the parent process. When the child process ends, its result code is discarded. The process ID is stored in the *codeTerminate* field of the RESULTCODES structure pointed to by *pRes* .

2 EXEC_ASYNCRESULT
Execution is asynchronous to the parent process. When the child process ends, its result code is saved for examination by a DosWaitChild request. The process ID is stored in the *codeTerminate* field of the RESULTCODES structure pointed to by *pRes* .

3 EXEC_TRACE
Execution is the same as if EXEC_ASYNCRESULT were specified for *execFlag* . Debugging conditions are present for the child process.

4 EXEC_BACKGROUND
Execution is asynchronous to and detached from the parent-process session. When the detached process starts, it is not affected by the ending of the parent process. The detached process is treated as an orphan of the parent process.

A program executed with this option runs in the background, and should not require any input from the keyboard or output to the screen other than VioPopups. It should not issue any console I/O calls (VIO, KBD, or MOU functions).

5 EXEC_LOAD
The program is loaded into storage and made ready to execute, but is not executed until the session manager dispatches the threads belonging to the process.

6 EXEC_ASYNCRESULTDB
Execution is the same as if EXEC_ASYNCRESULT were specified for *execFlag* , with the addition of debugging conditions being present for the child process and any of its descendants. In this way, it is possible to debug even detached and synchronous processes.

Some memory is consumed for uncollected result codes. Issue DosWaitChild to release this memory. If result codes are not collected, then EXEC_SYNC or EXEC_ASYNC should be used for *execFlag* .

DosExecPgm Parameter - pArg

pArg (PSZ) - input
Address of the ASCIIZ argument strings passed to the program.

These strings represent command parameters, which are copied to the environment segment of the new process.

The convention used by CMD.EXE is that the first of these strings is the program name (as entered from the command prompt or found in a batch file), and the second string consists of the parameters for the program. The second ASCIIZ string is followed by an additional byte of zeros. A value of zero for the address of *pArg* means that no arguments are to be passed to the program.

DosExecPgm Parameter - pEnv

pEnv (PSZ) - input
Address of the ASCIIZ environment strings passed to the program.

These strings represent environment variables and their current values. An environment string has the following form:

```
variable=value
```

The last ASCIIZ environment string must be followed by an additional byte of zeros.

A value of 0 for the address of *pEnv* results in the new process' inheriting the environment of its parent process.

When the new process is given control, it receives:

- A pointer to its environment segment
- The fully qualified file specification of the executable file
- A copy of the argument strings.

A coded example of this follows:

```
eo:   ASCIIZ string 1  ; environment string 1
      ASCIIZ string 2  ; environment string 2
.
.
.
      ASCIIZ string n  ; environment string n
      Byte of 0
.
.
.
po:   ASCIIZ           ; string of file name
```

```
                        ; of program to run.
.
.
.
ao:   ASCIIZ            ; argument string 1
                        ; (name of program being started
                        ; for the case of CMD.EXE)
      ASCIIZ            ; argument string 2
                        ; (program parameters following
                        ; program name for the case of
                        ; CMD.EXE)
      Byte of 0
```

The beginning of the environment segment is "eo", and "ao" is the offset of the first argument string in that segment. The offset to the command line, "ao", is passed to the program on the stack at SS:[ESP+16].

The environment strings typically have the form: parameter = value

A value of zero for *pEnv* causes the newly created process to inherit the parent's environment unchanged.

DosExecPgm Parameter - pRes

pRes (PRESULTCODES) - output
Pointer to the RESULTCODES structure where the process ID, or the termination code and the result code indicating the reason for ending the child process is returned.

This structure also is used by a DosWaitChild request, which waits for an asynchronous child process to end.

DosExecPgm Parameter - pName

pName (PSZ) - input
Address of the name of the file that contains the program to be executed.

When the environment is passed to the target program, this name is copied into "po" in the environment description shown above.

If the string appears to be a fully qualified file specification (that is, it contains a ":" or a "\" in the second position), then the file name must include the extension, and the program is loaded from the indicated drive:directory.

If the string is not a fully qualified path, the current directory is searched. If the file name is not found in the current directory, each drive:directory specification in the PATH defined in the current-process environment is searched for this file. Note that any extension (.XXX) is acceptable for the executable file being loaded.

DosExecPgm Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosExecPgm returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED

8	ERROR_NOT_ENOUGH_MEMORY
10	ERROR_BAD_ENVIRONMENT
11	ERROR_BAD_FORMAT
13	ERROR_INVALID_DATA
26	ERROR_NOT_DOS_DISK
31	ERROR_GEN_FAILURE
32	ERROR_SHARING_VIOLATION
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
89	ERROR_NO_PROC_SLOTS
95	ERROR_INTERRUPT
108	ERROR_DRIVE_LOCKED
127	ERROR_PROC_NOT_FOUND
182	ERROR_INVALID_ORDINAL
190	ERROR_INVALID_MODULETYPE
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID
195	ERROR_INVALID_MINALLOCSIZE
196	ERROR_DYNLINK_FROM_INVALID_RING

For a full list of error codes, see Errors.

DosExecPgm - Parameters

pObjname (PCHAR) - output
Address of the buffer in which the name of the object that contributed to the failure of DosExecPgm is returned.

cbObjname (LONG) - input
Length, in bytes, of the buffer described by *pObjname* .

execFlag (ULONG) - input
Flag indicating how the program runs in relation to the requester, and whether execution is under conditions for debugging.

The values of this field are shown in the following list:

0 EXEC_SYNC
Execution is synchronous to the parent process. The termination code and result code are stored in the RESULTCODES structure pointed to by *pRes* .

1 EXEC_ASYNC
Execution is asynchronous to the parent process. When the child process ends, its result code is discarded. The process ID is stored in the *codeTerminate* field of the RESULTCODES structure pointed to by *pRes* .

2 EXEC_ASYNCRESULT
Execution is asynchronous to the parent process. When the child process ends, its result code is saved for examination by a DosWaitChild request. The process ID is stored in the *codeTerminate* field of the RESULTCODES structure pointed to by *pRes* .

3 EXEC_TRACE
Execution is the same as if EXEC_ASYNCRESULT were specified for *execFlag* . Debugging conditions are present for the child process.

4 EXEC_BACKGROUND
Execution is asynchronous to and detached from the parent-process session. When the detached process starts, it is not affected by the ending of the parent process. The detached process is treated as an orphan of the parent process.

A program executed with this option runs in the background, and should not require any input from the keyboard or output to the screen other than VioPopups. It should not issue any console I/O calls (VIO, KBD, or MOU functions).

5 EXEC_LOAD
The program is loaded into storage and made ready to execute, but is not executed until the session manager dispatches the threads belonging to the process.

6 EXEC_ASYNCRESULTDB
Execution is the same as if EXEC_ASYNCRESULT were specified for *execFlag* , with the addition of debugging conditions being present for the child process and any of its descendants. In this way, it is possible to debug even detached and synchronous processes.

Some memory is consumed for uncollected result codes. Issue DosWaitChild to release this memory. If result codes are not collected, then EXEC_SYNC or EXEC_ASYNC should be used for *execFlag* .

pArg (PSZ) - input
Address of the ASCIIZ argument strings passed to the program.

These strings represent command parameters, which are copied to the environment segment of the new process.

The convention used by CMD.EXE is that the first of these strings is the program name (as entered from the command prompt or found in a batch file), and the second string consists of the parameters for the program. The second ASCIIZ string is followed by an additional byte of zeros. A value of zero for the address of *pArg* means that no arguments are to be passed to the program.

pEnv (PSZ) - input
Address of the ASCIIZ environment strings passed to the program.

These strings represent environment variables and their current values. An environment string has the following form:

```
variable=value
```

The last ASCIIZ environment string must be followed by an additional byte of zeros.

A value of 0 for the address of *pEnv* results in the new process' inheriting the environment of its parent process.

When the new process is given control, it receives:

- A pointer to its environment segment
- The fully qualified file specification of the executable file
- A copy of the argument strings.

A coded example of this follows:

```
eo:   ASCIIZ string 1  ; environment string 1
      ASCIIZ string 2  ; environment string 2
.
.
.
      ASCIIZ string n  ; environment string n
      Byte of 0
.
.
.
po:   ASCIIZ           ; string of file name
                       ; of program to run.
.
.
.
ao:   ASCIIZ           ; argument string 1
                       ; (name of program being started
                       ; for the case of CMD.EXE)
      ASCIIZ           ; argument string 2
                       ; (program parameters following
                       ; program name for the case of
                       ; CMD.EXE)
      Byte of 0
```

The beginning of the environment segment is "eo", and "ao" is the offset of the first argument string in that segment. The offset to the command line, "ao", is passed to the program on the stack at SS:[ESP+16].

The environment strings typically have the form: parameter = value

A value of zero for *pEnv* causes the newly created process to inherit the parent's environment unchanged.

pRes (PRESULTCODES) - output
Pointer to the RESULTCODES structure where the process ID, or the termination code and the result code indicating the reason for ending the child process is returned.

This structure also is used by a DosWaitChild request, which waits for an asynchronous child process to end.

pName (PSZ) - input
Address of the name of the file that contains the program to be executed.

When the environment is passed to the target program, this name is copied into "po" in the environment description shown above.

If the string appears to be a fully qualified file specification (that is, it contains a ":" or a "\" in the second position), then the file name must include the extension, and the program is loaded from the indicated drive:directory.

If the string is not a fully qualified path, the current directory is searched. If the file name is not found in the current directory, each drive:directory specification in the PATH defined in the current-process environment is searched for this file. Note that any extension (.XXX) is acceptable for the executable file being loaded.

ulrc (APIRET) - returns
Return Code.

DosExecPgm returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
10	ERROR_BAD_ENVIRONMENT
11	ERROR_BAD_FORMAT
13	ERROR_INVALID_DATA
26	ERROR_NOT_DOS_DISK
31	ERROR_GEN_FAILURE
32	ERROR_SHARING_VIOLATION
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
89	ERROR_NO_PROC_SLOTS
95	ERROR_INTERRUPT
108	ERROR_DRIVE_LOCKED
127	ERROR_PROC_NOT_FOUND
182	ERROR_INVALID_ORDINAL
190	ERROR_INVALID_MODULETYPE
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID
195	ERROR_INVALID_MINALLOCSIZE
196	ERROR_DYNLINK_FROM_INVALID_RING

For a full list of error codes, see Errors.

DosExecPgm - Remarks

DosExecPgm allows a program to request that another program execute as a child process.

The target program is located and loaded into storage (if necessary), a process is created for it and placed into execution. The execution of a child process can be synchronous or asynchronous to the execution of its parent process. If synchronous execution is indicated, the requesting thread waits for completion of the child process. Other threads in the requesting process may continue to run.

If asynchronous execution is indicated, DosExecPgm places the process ID of the started child process into the first doubleword of the *pRes* structure. If EXEC_ASYNCRESULT is specified for *execFlag* , the parent process can issue DosWaitChild (after DosExecPgm) to examine the result code returned when the child process ends. If the value of *execFlag* , is EXEC_ASYNC, the result code of the asynchronous child process is not returned to the parent process.

If synchronous execution is indicated, DosExecPgm places the termination code and result code into the *pRes* structure.

The new process is created with an address space separate and distinct from its parent; that is, a new linear address space is built for the process.

The new process inherits all file handles and pipes of its parent, although not necessarily with the same access rights:

- Files are inherited except for those opened with no inheritance indicated.
- Pipes are inherited.

A child process inherits file handles obtained by its parent process with DosOpen calls that indicated inheritance. The child process also inherits handles to pipes created by the parent process with DosCreatePipe. This means that the parent process has control over the meanings of standard input, output, and error. For example, the parent could write a series of records to a file, open the file as standard input, open a listing file as standard output, and then execute a sort program that takes its input from standard input and writes to standard output.

Because a child process can inherit handles, and a parent process controls the meanings of handles for standard I/O, the parent can duplicate inherited handles as handles for standard I/O. This permits the parent process and the child process to coordinate I/O to a pipe or file. For

example, a parent process can create two pipes with DosCreatePipe requests. It can issue DosDupHandle to redefine the read handle of one pipe as standard input (0x0000), and the write handle of the other pipe as standard output (0x0001). The child process uses the standard I/O handles, and the parent process uses the remaining read and write pipe handles. Thus, the child process reads what the parent process writes to one pipe, and the parent process reads what the child process writes to the other pipe.

When an inherited file handle is duplicated, the position of the file pointer is always the same for both handles, regardless of which handle repositions the file pointer.

An asynchronous process that was started because the value of *execFlag* was EXEC_TRACE or EXEC_ASYNCRESULTDB is provided a trace flag facility. This facility and the trace buffers provided by DosDebug enable a debugger to perform breakpoint debugging. DosStartSession provides additional debugging capabilities that allow a debugger to trace all processes associated with an application running in a child session, regardless of whether the process is started with DosExecPgm or DosStartSession.

A detached process is treated as an orphan of the parent process and runs in the background. Thus, it cannot make any VIO, KBD, or MOU calls, except from within a video pop-up requested by VioPopUp. To test whether a program is running detached, use the following method. Issue a video call, (for example, VioGetAnsi). If the call is not issued within a video pop-up and the process is detached, the video call returns error code ERROR_VIO_DETACHED.

You may use DosExecPgm to start a process that is of the same type as the starting process. Process types include Presentation Manager, text-windowed, and full-screen. You may not use DosExecPgm to start a process that is of a different type than the starting process.

You must use DosStartSession to start a new process from a process that is of a different type. For example, use DosStopSession to start a Presentation Manager process from a non-Presentation Manager process.

The following are the register conventions for 32-bit programs:

Register	Definition
EIP	Starting program entry address
ESP	Top of stack address
CS	Code selector for the base of the linear address space
DS, ES, SS	Data selector for the base of the linear address space
FS	Data selector for the thread information block
GS	0
EAX, EBX	0
ECX, EDX	0
ESI, EDI	0
EBP	0
[ESP+0]	Return address to the routine that calls DosExit
[ESP+4]	Module handle for the program module
[ESP+8]	0
[ESP+12]	Address of the environment data object
[ESP+16]	Offset to the command line in the environment data object.

DosExecPgm - Related Functions

Related Functions

- DosCreatePipe
- DosCreateThread
- DosExit
- DosKillProcess
- DosKillThread
- DosOpen
- DosWaitChild

DosExecPgm - Example Code

The fisrt example starts another program named "QSINFO.EXE" and waits for it to finish. It then displays the termination and return codes. Before running this program, compile and link the example code shown in DosExit with the name "QFSINFO.EXE".

The second example starts the program "CHKDSK" with the "C:" parameter, and waits until it finishes. Termination and return codes are displayed.

```
#define INCL_DOSPROCESS       /* Process and thread values */
```

```
#define INCL_DOS
#define INCL_DOSERRORS        /* DOS error values          */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR       LoadError[CCHMAXPATH] = {0};
PSZ         Args             = NULL;
PSZ         Envs             = NULL;
RESULTCODES ChildRC          = {0};
APIRET      rc               = NO_ERROR;  /* Return code */

  rc = DosExecPgm(LoadError,           /* Object name buffer           */
                  sizeof(LoadError),   /* Length of object name buffer */
                  EXEC_SYNC,           /* Asynchronous/Trace flags     */
                  Args,                /* Argument string              */
                  Envs,                /* Environment string           */
                  &ChildRC,            /* Termination codes            */
                  "qfsinfo.exe");      /* Program file name            */

  if (rc != NO_ERROR) {
     printf("DosExecPgm error: return code = %u\n",rc);
     return 1;
  } else {
     printf("DosExecPgm complete.  Termination Code: %u  Return Code: %u\n",
             ChildRC.codeTerminate,
             ChildRC.codeResult);  /* This is explicitly set by other pgm */
  } /* endif */

  return NO_ERROR;
  }



#define INCL_DOSPROCESS       /* Process and thread values */
#define INCL_DOSERRORS        /* DOS error values          */
#define INCL_DOS
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR       uchLoadError[CCHMAXPATH] = {0}; /* Error info from DosExecPgm */
RESULTCODES ChildRC          = {0};       /* Results from child process  */
PID         pidChild         = 0;         /* pid for child process       */
APIRET      rc               = NO_ERROR;  /* Return code                 */
UCHAR       uchEnvString[14]  = "ANYTHING=HERE\0";  /* Environment string */
UCHAR       uchArgString[14]  = "CHKDSK\0 C: \0";   /* Argument string    */

      /* The argument string consists of the following:
            - the name of the program (followed by a NULL)
            - any parameters supplied to the program (followed by 2 NULLs)
         Only 1 NULL is explicitly specified at the end of this string.
         ASCIIZ strings end with a NULL already, giving us 2 NULLs.       */

  rc = DosExecPgm(uchLoadError,          /* Object name buffer           */
                  sizeof(uchLoadError),  /* Length of object name buffer */
                  EXEC_ASYNCRESULT,      /* Asynchronous/Trace flags     */
                  uchArgString,          /* Argument string              */
                  uchEnvString,          /* Environment string           */
                  &ChildRC,              /* Returns pid of process on an
                                            asynchronous request         */
                  "CHKDSK.COM");         /* Program file name            */

  if (rc != NO_ERROR) {
     printf("DosExecPgm error: return code = %u\n",rc);
     return 1;
  }

  rc = DosWaitChild(DCWA_PROCESS,    /* Look at only the process specified */
                    DCWW_WAIT,       /* Wait until a child terminates      */
                    &ChildRC,        /* Termination codes returned         */
                    &pidChild,       /* pid of terminating process         */
                    ChildRC.codeTerminate);   /* Process (pid) to look at  */

  if (rc != NO_ERROR) {
     printf("DosWaitChild error: return code = %u\n",rc);
     return 1;
  } else {
     printf("Child complete.  Termination Code: %u  Return Code: %u\n",
             ChildRC.codeTerminate, ChildRC.codeResult);
  }
  return NO_ERROR;
```

```
}
```

DosExecPgm - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosExit

DosExit - Syntax

Ends the current thread or process.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG    action;  /*  Ends the process and all of its threads. */
ULONG    result;  /*  Program's completion code. */

DosExit(action, result);
```

DosExit Parameter - action

action (ULONG) - input
Ends the process and all of its threads.

The values of this field are as follows:

0	EXIT_THREAD The current thread ends.
1	EXIT_PROCESS All threads in the process end.

DosExit Parameter - result

result (ULONG) - input
Program's completion code.

It is passed to any thread that issues DosWaitChild for this process.

Note: Although *result* is a 32-bit parameter, the current release only permits the low 16 bits to be used. This means *result* must be <= 0xFFFF and implies that returning a 32-bit value (including NULL) is not possible.

DosExit - Return Value

Return Code.

DosExit returns no values.

DosExit - Parameters

action (ULONG) - input
Ends the process and all of its threads.

The values of this field are as follows:

0	EXIT_THREAD The current thread ends.
1	EXIT_PROCESS All threads in the process end.

result (ULONG) - input
Program's completion code.

It is passed to any thread that issues DosWaitChild for this process.

Note: Although *result* is a 32-bit parameter, the current release only permits the low 16 bits to be used. This means *result* must be <= 0xFFFF and implies that returning a 32-bit value (including NULL) is not possible.

Return Code.

DosExit returns no values.

DosExit - Remarks

DosExit is issued when a thread completes executing. The current thread or process ends.

DosExit allows a thread to terminate itself or be terminated by another thread in its process. If *action* is 0 and the specified thread is the last thread executing in the process, the process ends. If *action* is 1, the process ends.

The system can start threads on behalf of an application. Thus, if the intent of DosExit is to terminate the process, a value of 1 should be specified for *action* to end all the threads belonging to the process.

Do not end thread 1 without ending the process. Thread 1 is the initial thread of execution, not a thread started by a DosCreateThread request. When thread 1 ends, any monitors or signal processing routines set for this process also end. To avoid unpredictable results, DosExit should be issued with a value of 1 for *action* to ensure that the process ends.

When a process is ending, all but one thread is ended, and that thread executes routines whose addresses have been specified with DosExitList. After resources have been released by the exit list routines, this thread and all other resources owned by the process are released.

DosExit - Related Functions

Related Functions

- DosExecPgm
- DosExitList
- DosKillThread
- DosWaitChild

DosExit - Example Code

This example simulates the command "CHKDSK C: /F:O", but only shows the second half of the information. It returns the number of available allocation units to the initiating process. Some return code checking has been omitted for brevity.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>

int main (VOID) {
   ULONG   aulFSInfoBuf[40] = {0};         /* File system info buffer     */
   APIRET  rc               = NO_ERROR;    /* Return code                 */

   rc = DosQueryFSInfo(3L,                     /* Drive number 3 (C:)     */
                       FSIL_ALLOC,             /* Level 1 allocation info */
                       (PVOID)aulFSInfoBuf,    /* Buffer                  */
                       sizeof(aulFSInfoBuf));  /* Size of buffer          */

   if (rc != NO_ERROR) {
      printf("DosQueryFSInfo error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("%12ld bytes in each allocation unit.\n",
               aulFSInfoBuf[1] * (USHORT)aulFSInfoBuf[4]);
             /* (Sectors per allocation unit) * (Bytes per sector) */
      printf ("%12ld total allocation units.\n", aulFSInfoBuf[2]);
      printf ("%12ld available allocation units on disk.\n", aulFSInfoBuf[3]);
   }
   DosExit(EXIT_THREAD,aulFSInfoBuf[3]);  /* Return available allocation units
                                             to the initiating process      */
   return NO_ERROR;
}
```

DosExit - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosExitCritSec

DosExitCritSec - Syntax

Restores normal thread dispatching for the current process.

```
#define INCL_DOSPROCESS
#include <os2.h>

APIRET    ulrc;  /*  Return Code. */

ulrc = DosExitCritSec();
```

DosExitCritSec Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosExitCritSec returns one of the following values:

0	NO_ERROR
309	ERROR_INVALID_THREADID
485	ERROR_CRITSEC_UNDERFLOW

For a full list of error codes, see Errors.

DosExitCritSec - Parameters

ulrc (APIRET) - returns
Return Code.

DosExitCritSec returns one of the following values:

0	NO_ERROR
309	ERROR_INVALID_THREADID
485	ERROR_CRITSEC_UNDERFLOW

For a full list of error codes, see Errors.

DosExitCritSec - Remarks

DosExitCritSec is used following DosEnterCritSec to restore normal thread switching to the threads of a process.

A count is maintained of the number of times DosEnterCritSec is issued without a corresponding DosExitCritSec. The count is incremented by DosEnterCritSec, and decremented by DosExitCritSec. Normal thread dispatching is not restored until the count is zero.

The outstanding count is maintained in a word. If an underflow occurs (the count is decremented below zero), the count is set to zero, no operation is performed, and the request returns with ERROR_CRITSEC_UNDERFLOW.

ERROR_INVALID_THREADID is returned when an invalid attempt is made to exit a critical section of code in a signal handler or exception handler.

ERROR_INVALID_THREADID is also returned when a dynamic link library (DLL) routine incorrectly issues DosExitCritSec.

DosExitCritSec - Related Functions

Related Functions

- DosCreateThread
- DosEnterCritSec

DosExitCritSec - Example Code

This example shows how a thread can enter and exit a critical section of code.

```
#define INCL_DOSPROCESS      /* Process values */
#define INCL_DOSERRORS       /* Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID)
  {
  APIRET  rc = NO_ERROR;    /* Return code */

   rc = DosEnterCritSec();

   if (rc != NO_ERROR) {
       printf("DosEnterCritSec error: return code = %u\n",rc);
       return 1;
   }
     /*********************************************/
     /* Add critical section code here.  While this */
     /* code is running, all other threads are      */
     /* stopped.  CALL NO LIBRARY OR OS/2 FUNCTIONS */
     /* HERE UNLESS YOU KNOW THEY DO NOT REQUIRE    */
     /* ACTION BY ANOTHER THREAD IN THE PROCESS.    */
     /*********************************************/

   rc =  DosExitCritSec();

   if (rc != NO_ERROR) {
       printf("DosExitCritSec error: return code = %u\n",rc);
       return 1;
   }

   return NO_ERROR;
   }
```

DosExitCritSec - Topics

Select an item:
Syntax

Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosExitList

DosExitList - Syntax

Keeps a list of routines that run when the current process ends.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG          ordercode;  /*  DosExitList action and invocation order. */
PFNEXITLIST    pfn;        /*  The address of a routine to be run. */
APIRET         ulrc;       /*  Return Code. */

ulrc = DosExitList(ordercode, pfn);
```

DosExitList Parameter - ordercode

ordercode (ULONG) - input
DosExitList action and invocation order.

ordercode contains two one-byte fields in the low-order word. The high-order word is zero.

The low-order byte of the low-order word indicates which action DosExitList performs. This function can update the list of routines or transfer to the next address on the termination list at the completion of a routine. The values of the byte and their meanings are as follows:

1	EXLST_ADD Add an address to the termination list.
2	EXLST_REMOVE Remove an address from the termination list.
3	EXLST_EXIT When termination processing completes, transfer to the next address on the termination list.

The high-order byte of the low-order word indicates the invocation order. This value is valid only when the low-order byte is 1 (add an address). For the other low-order byte values, the high-order byte of the low-order word must be set to zero.

The invocation order indicates where the routine address is to be placed in an ordered list. This list determines the order in which the exit list routines are invoked. Routines with a value of 0 are invoked first, and routines with a value of 255 are invoked last. If more than one routine is added with the same invocation order value, the last routine to be added is invoked first. The following values are used by OS/2 components:

0x80 - 0x88	OS/2 Extended Edition Database Manager
0x90 - 0x98	OS/2 Extended Edition Communications Manager
0xA0 - 0xA8	OS/2 Presentation Manager
0xB0	OS/2 Keyboard (KBD) component

0xC0	OS/2 Video (VIO) component
0xD0	OS/2 Interprocess Communication (IPC) Queues component

DosExitList Parameter - pfn

pfn (PFNEXITLIST) - input
The address of a routine to be run.

DosExitList Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosExitList returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
8	ERROR_NOT_ENOUGH_MEMORY
13	ERROR_INVALID_DATA

For a full list of error codes, see Errors.

DosExitList - Parameters

ordercode (ULONG) - input
DosExitList action and invocation order.

ordercode contains two one-byte fields in the low-order word. The high-order word is zero.

The low-order byte of the low-order word indicates which action DosExitList performs. This function can update the list of routines or transfer to the next address on the termination list at the completion of a routine. The values of the byte and their meanings are as follows:

1	EXLST_ADD Add an address to the termination list.
2	EXLST_REMOVE Remove an address from the termination list.
3	EXLST_EXIT When termination processing completes, transfer to the next address on the termination list.

The high-order byte of the low-order word indicates the invocation order. This value is valid only when the low-order byte is 1 (add an address). For the other low-order byte values, the high-order byte of the low-order word must be set to zero.

The invocation order indicates where the routine address is to be placed in an ordered list. This list determines the order in which the exit list routines are invoked. Routines with a value of 0 are invoked first, and routines with a value of 255 are invoked last. If more than one routine is added with the same invocation order value, the last routine to be added is invoked first. The following values are used by OS/2 components:

0x80 - 0x88	OS/2 Extended Edition Database Manager
0x90 - 0x98	OS/2 Extended Edition Communications Manager
0xA0 - 0xA8	OS/2 Presentation Manager
0xB0	OS/2 Keyboard (KBD) component
0xC0	OS/2 Video (VIO) component

pfn (PFNEXITLIST) - input
The address of a routine to be run.

ulrc (APIRET) - returns
Return Code.

DosExitList returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
8	ERROR_NOT_ENOUGH_MEMORY
13	ERROR_INVALID_DATA

For a full list of error codes, see Errors.

DosExitList - Remarks

DosExitList is issued to define a routine that is to be given control when a process completes its execution. Multiple routines may be defined to receive control when a process is ending. For each process, the operating system maintains a list of addresses of defined exit list routines.

When the process is ending, the operating system transfers control to each address in this list. If there are multiple addresses in the list, they will each get control in numerical order by function invocation order, that is, low (0) will be first, and high (0xFF) will be last. In case of duplicate entries for the same value, the routines will be run in LIFO (last in, first out) order.

Library modules can issue DosExitList to free resources or to clear flags and semaphores in case the client process ends without notifying them.

Before transferring control to the routines in the termination list, the operating system resets the stack to its initial value. The routine must be in the address space of the ending process. The termination routine should perform its processing and then issue DosExitList with a value of 3 (EXLST_EXIT) for *ordercode* . The termination routine should be as short as possible.

Most system functions are allowed in an exit list routine. However, DosCreateThread and DosExecPgm are not.

An exit list routine must not call functions that have a better function order priority (that is, a lower value for *ordercode*) than itself. For example, an exit list routine with a function order value of 0x9A can use Presentation Services functions but not Communications Manager functions.

When the exit list routine receives control, the first parameter on the stack (located at ESP+4) contains a termination code that describes why the process ended. The values of the termination codes are as follows:

0 TC_EXIT
Normal exit

1 TC_HARDERROR
Hard-error halt

2 TC_TRAP
Trap operation for a 16-bit child process

3 TC_KILLPROCESS
Unintercepted DosKillProcess

4 TC_EXCEPTION
Exception operation for a 32-bit child process

When the exit list routine receives control, all system semaphores owned by the process have their ownership transferred to the thread that performs exit list processing. This allows the thread to request serialization semaphores without danger of blocking in case the semaphore was held by another thread in the process that has already ended.

Note: All exit list routines *must* be declared as VOID APIENTRY. This ensures the integrity of the stack.

DosExitList - Related Functions

Related Functions

- DosCreateThread
- DosExecPgm
- DosExit
- DosKillProcess
- DosKillThread

DosExitList - Example Code

This example adds an exit routine named "ExitRtn1" to the exit-list sequence. Routines in this sequence must use DosExitList instead of DosExit to end.

```
#define INCL_DOSPROCESS              /* Process and thread values */
#define INCL_DOSERRORS               /* DOS error values          */
#include <os2.h>
#include <stdio.h>

VOID APIENTRY ExitRtn1(VOID);        /* Our exit routine */

int main(VOID) {

   APIRET rc = NO_ERROR;

   rc = DosExitList(EXLST_ADD             /* Add to exit-list sequence     */
                  | 0x00002A00,           /* Invocation order is 42 (0x2A) */
                  (PFNEXITLIST) ExitRtn1); /* Specify added exit routine */

   if (rc != NO_ERROR) {
     printf("DosExitList error: return code = %u\n", rc);
     return 1;
   }

   printf("Routine main ends...\n");

   return NO_ERROR;
  }

     /* All exit list routines must be declared as VOID APIENTRY.
        This ensures the integrity of the stack.                  */

VOID APIENTRY ExitRtn1(VOID)
   {
   APIRET  erc = NO_ERROR;                    /* Return code */

   printf("... but ExitRtn1 runs last.\n");

          /* Might want to save data or close files here */

   erc = DosExitList(EXLST_EXIT,              /* Exit         */
                     (PFNEXITLIST) NULL);

   if (erc != NO_ERROR) {
     printf("DosExitList error: return code = %u\n", erc);
   }

   return;
   }
```

DosExitList - Topics

Select an item:
Syntax
Parameters

Returns
Remarks
Example Code
Related Functions
Glossary

DosExitMustComplete

DosExitMustComplete - Syntax

Provides exit from a section of code in which asynchronous exceptions are held.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PULONG    pulNesting;  /*  Pointer to the nesting level. */
APIRET    ulrc;        /*  Return Code. */

ulrc = DosExitMustComplete(pulNesting);
```

DosExitMustComplete Parameter - pulNesting

pulNesting (PULONG) - output
Pointer to the nesting level.

The nesting level is equal to the number of DosEnterMustComplete requests minus the number of DosExitMustComplete requests for the current thread.

DosExitMustComplete Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosExitMustComplete returns one of the following values:

0	NO_ERROR
300	ERROR_ALREADY_RESET

For a full list of error codes, see Errors.

DosExitMustComplete - Parameters

pulNesting (PULONG) - output
Pointer to the nesting level.

The nesting level is equal to the number of DosEnterMustComplete requests minus the number of DosExitMustComplete requests for the current thread.

ulrc (APIRET) - returns
Return Code.

DosExitMustComplete returns one of the following values:

0	NO_ERROR
300	ERROR_ALREADY_RESET

For a full list of error codes, see Errors.

DosExitMustComplete - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosExitMustComplete notifies the system that the calling thread is leaving a section of code in which any asynchronous exceptions (signals and asynchronous process terminations) that may have occurred were held, rather than being immediately delivered to the thread.

For a detailed list of the system exceptions, see System Exceptions.

DosExitMustComplete - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler
- DosUnwindException

DosExitMustComplete - Example Code

This example shows how a thread can notify the system to hold asynchronous exceptions during a section of code.

```
#define INCL_DOSEXCEPTIONS    /* Exception values */
#define INCL_DOSERRORS        /* Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID)
  {
  ULONG   ulNestLevel = 0;  /* Global variable tracking nesting
                               of DosEnterMustComplete calls     */
  APIRET  rc = NO_ERROR;    /* Return code                       */

   rc = DosEnterMustComplete(&ulNestLevel);

   if (rc != NO_ERROR) {
```

```
        printf("DosEnterMustComplete error: return code = %u\n",rc);
        return 1;
    } else {
        printf("ulNestLevel = %u\n",ulNestLevel);
    }

        /* ADD BLOCK OF CODE THAT MUST COMPLETE HERE... */

    rc = DosExitMustComplete(&ulNestLevel);

    if (rc != NO_ERROR) {
        printf("DosExitMustComplete error: return code = %u\n",rc);
        return 1;
    } else {
        printf("ulNestLevel = %u\n",ulNestLevel);
    }

    return NO_ERROR;
    }
```

DosExitMustComplete - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFindClose

DosFindClose - Syntax

Closes the handle to a find request; that is, ends a search.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HDIR      hDir;  /*  Directory search handle. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosFindClose(hDir);
```

DosFindClose Parameter - hDir

hDir (HDIR) - input

Directory search handle.

The handle previously associated with a DosFindFirst function by the system, or used with a DosFindNext directory search function.

DosFindClose Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFindClose returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosFindClose - Parameters

hDir (HDIR) - input
Directory search handle.

The handle previously associated with a DosFindFirst function by the system, or used with a DosFindNext directory search function.

ulrc (APIRET) - returns
Return Code.

DosFindClose returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosFindClose - Remarks

Once DosFindClose is issued, any subsequent DosFindNext issued for the closed handle (*hDir*) fails unless an intervening DosFindFirst specifying the handle is issued.

DosFindClose - Related Functions

Related Functions

- DosFindFirst
- DosFindNext

DosFindClose - Example Code

This example lists all the normal files that are in the directory from where the example is invoked.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main (VOID) {
   HDIR          hdirFindHandle = HDIR_CREATE;
   FILEFINDBUF3  FindBuffer     = {0};      /* Returned from FindFirst/Next */
   ULONG         ulResultBufLen = sizeof(FILEFINDBUF3);
   ULONG         ulFindCount    = 1;        /* Look for 1 file at a time    */
   APIRET        rc             = NO_ERROR; /* Return code                  */

   rc = DosFindFirst( "*.*",                /* File pattern - all files     */
                      &hdirFindHandle,      /* Directory search handle      */
                      FILE_NORMAL,          /* Search attribute             */
                      &FindBuffer,          /* Result buffer                */
                      ulResultBufLen,       /* Result buffer length         */
                      &ulFindCount,         /* Number of entries to find    */
                      FIL_STANDARD);        /* Return level 1 file info     */

   if (rc != NO_ERROR) {
      printf("DosFindFirst error: return code = %u\n",rc);
      return 1;
   } else {
      printf ("%s\n", FindBuffer.achName);   /* Print file name            */
   } /* endif */

   /* Keep finding the next file until there are no more files */
   while (rc != ERROR_NO_MORE_FILES) {
      ulFindCount = 1;                      /* Reset find count.            */

      rc = DosFindNext(hdirFindHandle,      /* Directory handle             */
                       &FindBuffer,         /* Result buffer                */
                       ulResultBufLen,      /* Result buffer length         */
                       &ulFindCount);       /* Number of entries to find    */

      if (rc != NO_ERROR && rc != ERROR_NO_MORE_FILES) {
         printf("DosFindNext error: return code = %u\n",rc);
         return 1;
      } else {
         printf ("%s\n", FindBuffer.achName);    /* Print file name */
      }
   } /* endwhile */

   rc = DosFindClose(hdirFindHandle);    /* Close our directory handle */
   if (rc != NO_ERROR) {
      printf("DosFindClose error: return code = %u\n",rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosFindClose - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFindFirst

DosFindFirst - Syntax

Finds the first file object or group of file objects whose names match the specification. The specification can include extended attributes (EA) associated with a file or directory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszFileSpec;  /*  Address of the ASCIIZ path name of the file or subdirectory to be found. */
PHDIR     phdir;        /*  Address of the handle associated with this DosFindFirst request. */
ULONG     flAttribute;  /*  Attribute value that determines the file objects to be searched for. */
PVOID     pfindbuf;     /*  Result buffer. */
ULONG     cbBuf;        /*  The length, in bytes, of pfindbuf. */
PULONG    pcFileNames;  /*  Pointer to the number of entries: */
ULONG     ulInfoLevel;  /*  The level of file information required. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosFindFirst(pszFileSpec, phdir, flAttribute,
         pfindbuf, cbBuf, pcFileNames, ulInfoLevel);
```

DosFindFirst Parameter - pszFileSpec

pszFileSpec (PSZ) - input
Address of the ASCIIZ path name of the file or subdirectory to be found.

The name component can contain global file name characters.

DosFindFirst Parameter - phdir

phdir (PHDIR) - in/out
Address of the handle associated with this DosFindFirst request.

The values that can be specified for the handle are shown in the following list:

HDIR_CREATE (0xFFFFFFFF)
The system allocates and returns a handle. Upon return to the caller, *phdir* contains the handle allocated by the system for use by the process. Allocating a handle for each search allows recursive searches to be done and provides better performance on lengthy searches. This is the recommended option to use.

HDIR_SYSTEM (0x00000001)
The system assigns the handle for standard output, which is always available to a process. This handle should never be closed and used only for short, non-recursive searches in a single threaded program.

The DosFindFirst handle is used with subsequent DosFindNext requests. Reuse of this handle in another DosFindFirst request closes the association with the previous DosFindFirst request, and opens a new association with the current DosFindFirst request.

DosFindFirst Parameter - flAttribute

flAttribute (ULONG) - input

Attribute value that determines the file objects to be searched for.

The bit values are shown in the following list:

31-14	Reserved; must be 0.
13	MUST_HAVE_ARCHIVED (0x00002000) Must-Have Archive bit; excludes files without the archive bit set if bit 13 is set to 1. Files may have the Archive bit set if bit 13 is set to 0.
12	MUST_HAVE_DIRECTORY (0x00001000) Must-Have Subdirectory bit; excludes files that are not subdirectories if bit 12 is set to 1. Files may have the Subdirectory bit set if bit 12 is set to 0.
11	Reserved; must be 0.
10	MUST_HAVE_SYSTEM (0x00000400) Must-Have System File bit; excludes nonsystem files if bit 10 is set to 1. Files may be system files if bit 10 is set to 0.
9	MUST_HAVE_HIDDEN (0x00000200) Must-Have Hidden File bit; excludes nonhidden files if bit 9 is set to 1. Files may be nonhidden if bit 9 is set to 0.
8	MUST_HAVE_READONLY (0x00000100) Must-Have Read-Only File bit; excludes writeable files if bit 8 is set to 1. Files may be read-only if bit 8 is set to 0.
7-6	Reserved; must be 0.
5	FILE_ARCHIVED (0x00000020) May-Have Archive bit; includes files with the Archive bit set if bit 5 is set to 1. Excludes files with the Archive bit set if bit 5 is set to 0.
4	FILE_DIRECTORY (0x00000010) May-Have Subdirectory bit; includes files that are subdirectories if bit 4 is set to 1. Excludes files that are subdirectories if bit 4 is set to 0.
3	Reserved; must be 0.
2	FILE_SYSTEM (0x00000004) May-Have System File bit; includes system files if bit 2 is set to 1. Excludes system files if bit 2 is set to 0.
1	FILE_HIDDEN (0x00000002) May-Have Hidden File bit; includes hidden files if bit 1 is set to 1. Excludes hidden files if bit 1 is set to 0.
0	FILE_READONLY (0x00000001) May-Have Read-Only File bit; includes read-only files if bit 0 is set to 1. Excludes read-only files if bit 0 is set to 0.

These bits may be set individually or in combination. For example, an attribute value of 0x00000024 (bits 5 and 2 set to 1) indicates searching for system files that have been archived.

Bits 8 through 13 are "Must-Have" flags. These allow you to obtain files that definitely have the given attributes. For example, if the Must-Have Subdirectory bit is set to 1, then all returned items are subdirectories.

If a Must-Have bit is set to 1, and the corresponding May-Have bit is set to 0, no items are returned for that attribute.

The attribute FILE_NORMAL (0x00000000) can be used to include files with any of the above bits set.

flAttribute cannot specify the volume label. Volume labels are queried using DosQueryFSInfo.

DosFindFirst Parameter - pfindbuf

pfindbuf (PVOID) - in/out
Result buffer.

The result buffer from DosFindFirst should be less than 64KB.

Address of the directory search structures for file object information Levels 1 through 3. The structure required for *pfindbuf* is dependent on the value specified for *ullnfoLevel*. The information returned reflects the most recent call to DosClose or DosResetBuffer.

For Level 1 File Information (*ullnfoLevel* == FIL_STANDARD) :
On output, *pfindbuf* contains the FILEFINDBUF3 data structure without the last two fields: *cchName* and *achName*. This is used without EAs.

The *oNextEntryOffset* field indicates the number of bytes from the beginning of the current structure to the beginning of the next structure. When this field is 0, the last structure has been reached.

For Level 2 File Information (*ullnfoLevel* == FIL_QUERYEASIZE) :
On output, *pfindbuf* contains the FILEFINDBUF4 data structure without the last two fields: *cchName* and *achName*. This is used with EAs.

The *cbList* field contains the size, in bytes, of the file's entire EA set on disk. You can use this field to calculate the maximum size of the buffer needed for Level 3 file information. The size of the buffer required to hold the entire EA set is less than or equal to twice the size of the EA set on disk.

For Level 3 File Information (*ullnfoLevel* == FIL_QUERYEASFROMLIST) :
On input, *pfindbuf* contains an EAOP2 data structure. *fpGEA2List* contains a pointer to a GEA2 list, which defines the attribute names whose values are to be returned. Entries in the GEA2 list must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry.

On output, *pfindbuf* contains a structure with a set of records, each aligned on a doubleword boundary. These records represent the directory entry and associated EAs for the matched file object. *pfindbuf* has the following format:

- The EAOP2 data structure, with the *fpFEA2List* pointer incorrect.

 The EAOP2 data structure occurs only once in the *pfindbuf* buffer. The rest of these records are repeated for the remainder of the file objects found.
- A FILEFINDBUF3 data structure without the last two fields: *cchName* and *achName*.
- A FEA2LIST data structure contained in and related to the FILEFINDBUF3 returned.
- Length of the name string of the file object (*cbName*)
- Name of the file object matched by the input pattern (*achName*)

Even if there is not enough room to hold all of the requested information, as for return code ERROR_BUFFER_OVERFLOW, the *cbList* field of the FEA2LIST data structure is valid if there is at least enough space to hold it.

When buffer overflow occurs, *cbList* contains the size on disk of the entire EA set for the file, even if only a subset of its attributes was requested. The size of the buffer required to hold the EA set is less than or equal to twice the size of the EA set on disk. If no error occurs, *cbList* includes the pad bytes (for doubleword alignment) between FEA2 structures in the list.

If a particular attribute is not attached to the object, *pfindbuf* has an FEA2 structure containing the name of the attribute, and the length value is 0.

The GEA2 list contained inside *pfindbuf* during a Level 3 DosFindFirst and DosFindNext call is not "read-only"; it is used by OS/2. When the function returns, the list is restored to it's original state, but inside the function, the list is manipulated by OS/2. This is of concern to a multithreaded application, where two different threads might use the same GEA2 list as input. If one thread calls DosFindFirst or DosFindNext while another thread is inside DosFindFirst or DosFindNext, the second thread will fail with a return code of ERROR_BAD_FORMAT.

DosFindFirst Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of *pfindbuf*.

DosFindFirst Parameter - pcFileNames

pcFileNames (PULONG) - in/out
Pointer to the number of entries:

Input	The address of the number of matching entries requested in *pfindbuf* .
Output	The number of entries placed into *pfindbuf* .

DosFindFirst Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
The level of file information required.

Possible values are:

1	FIL_STANDARD Level 1 file information (return standard file information).
2	FIL_QUERYEASIZE Level 2 file information (return full EA size).
3	FIL_QUERYEASFROMLIST Level 3 file information (return requested EA).

The structures described in *pfindbuf* indicate the information returned for each of these levels.

Regardless of the level specified, a DosFindFirst request (and an associated DosFindNext) request on a handle returned by DosFindFirst) always includes Level 1 information as part of the information that is returned; however, when Level 1 information is specifically requested, and *flAttribute* specifies hidden files, system files, or subdirectory files, an inclusive search is made. That is, all normal file entries plus all entries matching any specified attributes are returned. Normal files are files without any mode bits set. They may be read from or written to.

DosFindFirst Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFindFirst returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
6	ERROR_INVALID_HANDLE
18	ERROR_NO_MORE_FILES
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
111	ERROR_BUFFER_OVERFLOW
113	ERROR_NO_MORE_SEARCH_HANDLES
206	ERROR_FILENAME_EXCED_RANGE
208	ERROR_META_EXPANSION_TOO_LONG
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT
275	ERROR_EAS_DIDNT_FIT

For a full list of error codes, see Errors.

DosFindFirst - Parameters

pszFileSpec (PSZ) - input
Address of the ASCIIZ path name of the file or subdirectory to be found.

The name component can contain global file name characters.

phdir (PHDIR) - in/out
Address of the handle associated with this DosFindFirst request.

The values that can be specified for the handle are shown in the following list:

HDIR_CREATE (0xFFFFFFFF)
The system allocates and returns a handle. Upon return to the caller, *phdir* contains the handle allocated by the system for use by the process. This is the recommended option to use.

HDIR_SYSTEM (0x00000001)
The system assigns the handle for standard output, which is always available to a process.

The DosFindFirst handle is used with subsequent DosFindNext requests. Reuse of this handle in another DosFindFirst request closes the association with the previous DosFindFirst request, and opens a new association with the current DosFindFirst request.

flAttribute (ULONG) - input
Attribute value that determines the file objects to be searched for.

The bit values are shown in the following list:

31-14	Reserved; must be 0.
13	MUST_HAVE_ARCHIVED (0x00002000) Must-Have Archive bit; excludes files without the archive bit set if bit 13 is set to 1. Files may have the Archive bit set if bit 13 is set to 0.
12	MUST_HAVE_DIRECTORY (0x00001000) Must-Have Subdirectory bit; excludes files that are not subdirectories if bit 12 is set to 1. Files may have the Subdirectory bit set if bit 12 is set to 0.
11	Reserved; must be 0.
10	MUST_HAVE_SYSTEM (0x00000400) Must-Have System File bit; excludes nonsystem files if bit 10 is set to 1. Files may be system files if bit 10 is set to 0.
9	MUST_HAVE_HIDDEN (0x00000200) Must-Have Hidden File bit; excludes nonhidden files if bit 9 is set to 1. Files may be nonhidden if bit 9 is set to 0.
8	MUST_HAVE_READONLY (0x00000100) Must-Have Read-Only File bit; excludes writeable files if bit 8 is set to 1. Files may be read-only if bit 8 is set to 0.
7-6	Reserved; must be 0.
5	FILE_ARCHIVED (0x00000020) May-Have Archive bit; includes files with the Archive bit set if bit 5 is set to 1. Excludes files with the Archive bit set if bit 5 is set to 0.
4	FILE_DIRECTORY (0x00000010) May-Have Subdirectory bit; includes files that are subdirectories if bit 4 is set to 1. Excludes files that are subdirectories if bit 4 is set to 0.
3	Reserved; must be 0.
2	FILE_SYSTEM (0x00000004) May-Have System File bit; includes system files if bit 2 is set to 1. Excludes system files if bit 2 is set to 0.

1	FILE_HIDDEN (0x00000002) May-Have Hidden File bit; includes hidden files if bit 1 is set to 1. Excludes hidden files if bit 1 is set to 0.
0	FILE_READONLY (0x00000001) May-Have Read-Only File bit; includes read-only files if bit 0 is set to 1. Excludes read-only files if bit 0 is set to 0.

These bits may be set individually or in combination. For example, an attribute value of 0x00000024 (bits 5 and 2 set to 1) indicates searching for system files that have been archived.

Bits 8 through 13 are "Must-Have" flags. These allow you to obtain files that definitely have the given attributes. For example, if the Must-Have Subdirectory bit is set to 1, then all returned items are subdirectories.

If a Must-Have bit is set to 1, and the corresponding May-Have bit is set to 0, no items are returned for that attribute.

The attribute FILE_NORMAL (0x00000000) can be used to include files with any of the above bits set.

flAttribute cannot specify the volume label. Volume labels are queried using DosQueryFSInfo.

pfindbuf (PVOID) - in/out
Result buffer.

The result buffer from DosFindFirst should be less than 64KB.

Address of the directory search structures for file object information Levels 1 through 3. The structure required for *pfindbuf* is dependent on the value specified for *ulInfoLevel*. The information returned reflects the most recent call to DosClose or DosResetBuffer.

For Level 1 File Information (*ulInfoLevel* == FIL_STANDARD) :
On output, *pfindbuf* contains the FILEFINDBUF3 data structure without the last two fields: *cchName* and *achName*. This is used without EAs.

The *oNextEntryOffset* field indicates the number of bytes from the beginning of the current structure to the beginning of the next structure. When this field is 0, the last structure has been reached.

For Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE) :
On output, *pfindbuf* contains the FILEFINDBUF4 data structure without the last two fields: *cchName* and *achName*. This is used with EAs.

The *cbList* field contains the size, in bytes, of the file's entire EA set on disk. You can use this field to calculate the maximum size of the buffer needed for Level 3 file information. The size of the buffer required to hold the entire EA set is less than or equal to twice the size of the EA set on disk.

For Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST) :
On input, *pfindbuf* contains an EAOP2 data structure. *fpGEA2List* contains a pointer to a GEA2 list, which defines the attribute names whose values are to be returned. Entries in the GEA2 list must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry.

On output, *pfindbuf* contains a structure with a set of records, each aligned on a doubleword boundary. These records represent the directory entry and associated EAs for the matched file object. *pfindbuf* has the following format:

- The EAOP2 data structure, with the *fpFEA2List* pointer incorrect.

 The EAOP2 data structure occurs only once in the *pfindbuf* buffer. The rest of these records are repeated for the remainder of the file objects found.
- A FILEFINDBUF3 data structure without the last two fields: *cchName* and *achName*.
- A FEA2LIST data structure contained in and related to the FILEFINDBUF3 returned.
- Length of the name string of the file object (*cbName*)
- Name of the file object matched by the input pattern (*achName*)

Even if there is not enough room to hold all of the requested information, as for return code ERROR_BUFFER_OVERFLOW, the *cbList* field of the FEA2LIST data structure is valid if there is at least enough space to hold it.

When buffer overflow occurs, *cbList* contains the size on disk of the entire EA set for the file, even if only a subset of its attributes was requested. The size of the buffer required to hold the EA set is less than or equal to twice the size of the EA set on disk. If no error occurs, *cbList* includes the pad bytes (for doubleword alignment) between FEA2 structures in the list.

If a particular attribute is not attached to the object, *pfindbuf* has an FEA2 structure containing the name of the attribute, and the length value is 0.

The GEA2 list contained inside *pfindbuf* during a Level 3 DosFindFirst and DosFindNext call is not "read-only"; it is used by OS/2. When the function returns, the list is restored to it's original state, but inside the function, the list is manipulated by OS/2. This is of concern to a multithreaded application, where two different threads might use the same GEA2 list as input. If one thread calls DosFindFirst or DosFindNext while another thread is inside DosFindFirst or DosFindNext, the second thread will fail with a return code of ERROR_BAD_FORMAT.

cbBuf (ULONG) - input
The length, in bytes, of *pfindbuf*.

pcFileNames (PULONG) - in/out
Pointer to the number of entries:

Input	The address of the number of matching entries requested in *pfindbuf*.
Output	The number of entries placed into *pfindbuf*.

ulInfoLevel (ULONG) - input
The level of file information required.

Possible values are:

1	FIL_STANDARD Level 1 file information (return standard file information).
2	FIL_QUERYEASIZE Level 2 file information (return full EA size).
3	FIL_QUERYEASFROMLIST Level 3 file information (return requested EA).

The structures described in *pfindbuf* indicate the information returned for each of these levels.

Regardless of the level specified, a DosFindFirst request (and an associated DosFindNext) request on a handle returned by DosFindFirst) always includes Level 1 information as part of the information that is returned; however, when Level 1 information is specifically requested, and *flAttribute* specifies hidden files, system files, or subdirectory files, an inclusive search is made. That is, all normal file entries plus all entries matching any specified attributes are returned. Normal files are files without any mode bits set. They may be read from or written to.

ulrc (APIRET) - returns
Return Code.

DosFindFirst returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
6	ERROR_INVALID_HANDLE
18	ERROR_NO_MORE_FILES
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
111	ERROR_BUFFER_OVERFLOW
113	ERROR_NO_MORE_SEARCH_HANDLES
206	ERROR_FILENAME_EXCED_RANGE
208	ERROR_META_EXPANSION_TOO_LONG
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT
275	ERROR_EAS_DIDNT_FIT

For a full list of error codes, see Errors.

DosFindFirst - Remarks

The result buffer from DosFindFirst should be less than 64KB.

DosFindFirst returns directory entries (up to the number requested in *pcFileNames*) and extended-attribute information for as many files or subdirectories whose names, attributes, and EAs match the specification, and whose information fits in *pfindbuf*. On output, *pcFileNames* contains the actual number of directory entries returned.

The file name pointed to by *pszFileSpec* can contain global file-name characters.

DosFindNext uses the directory handle associated with DosFindFirst to continue the search started by the DosFindFirst request.

Any nonzero return code, except ERROR_EAS_DIDNT_FIT, indicates that no handle has been allocated. This includes such nonerror indicators as ERROR_NO_MORE_FILES.

For ERROR_EAS_DIDNT_FIT, a search handle is returned, and a subsequent call to DosFindNext gets the next matching entry in the directory. You can use DosQueryPathInfo to retrieve the EAs for the matching entry by using the same EA arguments used for the DosFindFirst call, and the name that was returned by DosFindFirst.

For ERROR_EAS_DIDNT_FIT, only information for the first matching entry is returned. This entry is the one whose extended attributes did not fit in the buffer. The information returned is in the format of that returned for information Level 2. No further entries are returned in the buffer, even if they could fit in the remaining space.

The GEA2 list contained inside *pfindbuf* during a Level 3 DosFindFirst and DosFindNext call is not "read-only", it is used by OS/2. When the function returns, the list is restored to it's original state, but inside the function, the list is manipulated by OS/2. This is of concern to a multithreaded application, where two different threads might use the same GEA2 list as input. If one thread calls DosFindFirst or DosFindNext while another thread is inside DosFindFirst or DosFindNext, the second thread will fail with a return code of ERROR_BAD_FORMAT.

DosFindFirst - Related Functions

Related Functions

- DosClose
- DosFindClose
- DosFindNext
- DosQueryFileInfo
- DosQueryPathInfo
- DosQuerySysInfo
- DosResetBuffer
- DosSearchPath
- DosSetFileInfo
- DosSetPathInfo

DosFindFirst - Example Code

This example lists all the normal files that are in the directory from where the example is invoked.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main (VOID) {
   HDIR          hdirFindHandle = HDIR_CREATE;
   FILEFINDBUF3  FindBuffer     = {0};      /* Returned from FindFirst/Next */
   ULONG         ulResultBufLen = sizeof(FILEFINDBUF3);
   ULONG         ulFindCount    = 1;        /* Look for 1 file at a time    */
   APIRET        rc             = NO_ERROR; /* Return code                  */

   rc = DosFindFirst( "*.*",               /* File pattern - all files     */
                      &hdirFindHandle,     /* Directory search handle      */
                      FILE_NORMAL,         /* Search attribute             */
                      &FindBuffer,         /* Result buffer                */
                      ulResultBufLen,      /* Result buffer length         */
                      &ulFindCount,        /* Number of entries to find    */
                      FIL_STANDARD);       /* Return Level 1 file info     */

   if (rc != NO_ERROR) {
      printf("DosFindFirst error: return code = %u\n",rc);
      return 1;
   } else {
      printf ("%s\n", FindBuffer.achName);   /* Print file name            */
   } /* endif */

   /* Keep finding the next file until there are no more files */
```

```
   while (rc != ERROR_NO_MORE_FILES) {
      ulFindCount = 1;                     /* Reset find count.            */

      rc = DosFindNext(hdirFindHandle,     /* Directory handle             */
                       &FindBuffer,        /* Result buffer                */
                       ulResultBufLen,     /* Result buffer length         */
                       &ulFindCount);      /* Number of entries to find    */

      if (rc != NO_ERROR && rc != ERROR_NO_MORE_FILES) {
         printf("DosFindNext error: return code = %u\n",rc);
         return 1;
      } else {
         printf ("%s\n", FindBuffer.achName);    /* Print file name */
      }
   } /* endwhile */

   rc = DosFindClose(hdirFindHandle);    /* Close our directory handle */
   if (rc != NO_ERROR) {
      printf("DosFindClose error: return code = %u\n",rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosFindFirst - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFindNext

DosFindNext - Syntax

Finds the next set of file objects whose names match the specification in a previous call to DosFindFirst or DosFindNext.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HDIR      hDir;         /*  The handle of the directory. */
PVOID     pfindbuf;     /*  The address of the directory search information structure. */
ULONG     cbfindbuf;    /*  The length, in bytes, of pfindbuf. */
PULONG    pcFilenames;  /*  Pointer to the number of entries. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosFindNext(hDir, pfindbuf, cbfindbuf,
         pcFilenames);
```

DosFindNext Parameter - hDir

hDir (HDIR) - input
The handle of the directory.

DosFindNext Parameter - pfindbuf

pfindbuf (PVOID) - in/out
The address of the directory search information structure.

The information returned reflects the most recent call to DosClose or DosResetBuffer.

For the continuation of a Level 3 (FIL_QUERYEASFROMLIST) File Informationsearch, this buffer should contain input in the same format as a Level 3 File Information search by DosFindFirst.

See the description of the *pfindbuf* parameter in DosFindFirst for information about the output data that the file system driver places into this buffer.

DosFindNext Parameter - cbfindbuf

cbfindbuf (ULONG) - input
The length, in bytes, of *pfindbuf*.

DosFindNext Parameter - pcFilenames

pcFilenames (PULONG) - in/out
Pointer to the number of entries.

Input	The address of the number of matching entries requested in *pfindbuf*.
Output	The number of entries placed into *pfindbuf*.

DosFindNext Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFindNext returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
18	ERROR_NO_MORE_FILES

26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
275	ERROR_EAS_DIDNT_FIT

For a full list of error codes, see Errors.

DosFindNext - Parameters

hDir (HDIR) - input
The handle of the directory.

pfindbuf (PVOID) - in/out
The address of the directory search information structure.

The information returned reflects the most recent call to DosClose or DosResetBuffer.

For the continuation of a Level 3 (FIL_QUERYEASFROMLIST) File Informationsearch, this buffer should contain input in the same format as a Level 3 File Information search by DosFindFirst.

See the description of the *pfindbuf* parameter in DosFindFirst for information about the output data that the file system driver places into this buffer.

cbfindbuf (ULONG) - input
The length, in bytes, of *pfindbuf*.

pcFilenames (PULONG) - in/out
Pointer to the number of entries.

Input	The address of the number of matching entries requested in *pfindbuf*.
Output	The number of entries placed into *pfindbuf*.

ulrc (APIRET) - returns
Return Code.

DosFindNext returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
18	ERROR_NO_MORE_FILES
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
275	ERROR_EAS_DIDNT_FIT

For a full list of error codes, see Errors.

DosFindNext - Remarks

If ERROR_BUFFER_OVERFLOW is returned, further calls to DosFindNext start the search from the same entry.

If ERROR_EAS_DIDNT_FIT is returned, the buffer is too small to hold the extended attributes (EAs) for the first matching entry being returned. A subsequent call to DosFindNext gets the next matching entry. This enables the search to continue if the extended attributes being returned are too large for the buffer. You can use DosQueryPathInfo to retrieve the extended attributes for the matching entry by using the same EA arguments used for the call to DosFindFirst, and the name that was returned by DosFindFirst,

In the case of ERROR_EAS_DIDNT_FIT, only information for the first matching entry is returned. This is the entry whose extended attributes did not fit in the buffer. The information returned is in the format of Level 2 (FIL_QUERYEASIZE) File Information (FILEFINDBUF4). No further entries are returned in the buffer, even if they could fit in the remaining space.

DosFindNext - Related Functions

Related Functions

- DosClose
- DosFindClose
- DosFindFirst
- DosQueryFileInfo
- DosQueryPathInfo
- DosQuerySysInfo
- DosResetBuffer
- DosSearchPath
- DosSetFileInfo
- DosSetPathInfo

DosFindNext - Example Code

This example lists all the normal files that are in the directory from where the example is invoked.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main (VOID) {
   HDIR          hdirFindHandle = HDIR_CREATE;
   FILEFINDBUF3  FindBuffer     = {0};      /* Returned from FindFirst/Next */
   ULONG         ulResultBufLen = sizeof(FILEFINDBUF3);
   ULONG         ulFindCount    = 1;        /* Look for 1 file at a time    */
   APIRET        rc             = NO_ERROR; /* Return code                  */

   rc = DosFindFirst( "*.*",               /* File pattern - all files     */
                      &hdirFindHandle,     /* Directory search handle      */
                      FILE_NORMAL,         /* Search attribute             */
                      &FindBuffer,         /* Result buffer                */
                      ulResultBufLen,      /* Result buffer length         */
                      &ulFindCount,        /* Number of entries to find    */
                      FIL_STANDARD);       /* Return level 1 file info     */

   if (rc != NO_ERROR) {
      printf("DosFindFirst error: return code = %u\n",rc);
      return 1;
   } else {
      printf ("%s\n", FindBuffer.achName);   /* Print file name            */
   } /* endif */

   /* Keep finding the next file until there are no more files */
   while (rc != ERROR_NO_MORE_FILES) {
      ulFindCount = 1;                     /* Reset find count.            */

      rc = DosFindNext(hdirFindHandle,     /* Directory handle             */
                       &FindBuffer,        /* Result buffer                */
                       ulResultBufLen,     /* Result buffer length         */
                       &ulFindCount);      /* Number of entries to find    */

      if (rc != NO_ERROR && rc != ERROR_NO_MORE_FILES) {
         printf("DosFindNext error: return code = %u\n",rc);
         return 1;
      } else {
         printf ("%s\n", FindBuffer.achName);    /* Print file name */
      }
   } /* endwhile */

   rc = DosFindClose(hdirFindHandle);    /* Close our directory handle */
   if (rc != NO_ERROR) {
      printf("DosFindClose error: return code = %u\n",rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosFindNext - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosForceDelete

DosForceDelete - Syntax

Removes a file name from a directory. The deleted file is not recoverable.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszFile;  /*  Address of the name of the file to be deleted. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosForceDelete(pszFile);
```

DosForceDelete Parameter - pszFile

pszFile (PSZ) - input
Address of the name of the file to be deleted.

DosForceDelete Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosForceDelete returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND

3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosForceDelete - Parameters

pszFile (PSZ) - input
Address of the name of the file to be deleted.

ulrc (APIRET) - returns
Return Code.

DosForceDelete returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosForceDelete - Remarks

Global file-name characters are not permitted in the name of the file to be deleted.

Read-only files cannot be deleted by DosForceDelete. To delete a read-only file, you must first issue DosSetFileInfo to change the file's read-only attribute to zero, then delete the file.

The deleted file cannot be recovered with the UNDELETE command. You may want to issue DosForceDelete to delete a temporary file that you would not want to recover.

DosForceDelete cannot be used to delete directories. Issue DosDeleteDir to delete a directory.

DosForceDelete - Related Functions

Related Functions

- DosDelete
- DosDeleteDir
- DosSetFileInfo

DosForceDelete - Example Code

This example creates a read-only file named "DOSFDEL.DAT", it then changes the file attributes. DosForceDelete is used to delete this file so it is not possible to restore it using UNDELETE.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 UCHAR       uchFileName[]   = "DOSFDEL.DAT";   /* File we want to delete    */
 HFILE       fhDelFile       = 0;               /* File handle from DosOpen  */
 FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
 ULONG       ulBufferSize    = sizeof(FILESTATUS3); /* File info buffer size */
 ULONG       ulOpenAction    = 0;                /* Action taken by DosOpen */
 APIRET      rc              = NO_ERROR;         /* Return code             */

                /* Create a read-only file */

  rc = DosOpen(uchFileName, &fhDelFile,
               &ulOpenAction, 10L, FILE_READONLY,
               OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
               OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L);
  if (rc != NO_ERROR) {
     printf("DosOpen error: return code = %u\n", rc);
     return 1;       }

  rc = DosQueryFileInfo(fhDelFile, FIL_STANDARD,
                        &fsts3FileInfo, ulBufferSize);  /* Get standard info */
  if (rc != NO_ERROR) {
      printf("DosQueryFileInfo error: return code = %u\n", rc);
      return 1;
  } else { printf("File %s created read-only.\n",uchFileName); }

    fsts3FileInfo.attrFile  = FILE_NORMAL;
    rc = DosSetFileInfo(fhDelFile, FIL_STANDARD,
                        &fsts3FileInfo, ulBufferSize);
    if (rc != NO_ERROR) {
        printf("DosSetFileInfo error: return code = %u\n", rc);
        return 1;
    }

    rc = DosClose(fhDelFile);
    /* should check (rc != NO_ERROR) here... */

           /* Delete the file */

    rc = DosForceDelete(uchFileName);
    if (rc != NO_ERROR) {
        printf("DosForceDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    } /* endif */

   return NO_ERROR;
}
```

DosForceDelete - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFreeMem

DosFreeMem - Syntax

Frees a private or shared memory object from the virtual-address space of the process.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pb;    /*  The base virtual address of the private or shared memory object whose reference is to be fr
APIRET    ulrc;  /*  Return Code. */

ulrc = DosFreeMem(pb);
```

DosFreeMem Parameter - pb

pb (PVOID) - input
The base virtual address of the private or shared memory object whose reference is to be freed.

DosFreeMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFreeMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
95	ERROR_INTERRUPT
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosFreeMem - Parameters

pb (PVOID) - input
The base virtual address of the private or shared memory object whose reference is to be freed.

ulrc (APIRET) - returns
Return Code.

DosFreeMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
95	ERROR_INTERRUPT
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosFreeMem - Remarks

DosFreeMem releases a previously allocated private or shared memory object from the virtual-address space of the subject process. The released pages are given an access protection of no access.

Freeing a shared memory object decrements the reference count for the associated object. If the resulting count is zero (that is, no other references to the shared memory object exist throughout the system), then the object is deleted. The deletion of the shared memory object releases the backing storage for the committed pages within the object.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosFreeMem - Related Functions

Related Functions

- DosAllocMem
- DosAllocSharedMem

DosFreeMem - Example Code

This example allocates, queries, commits, uses, and finally frees memory.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main (VOID)
{
   PVOID  MyObject    = NULL;      /* Pointer to memory object            */
   ULONG  ulObjSize   = 0;         /* Size of memory object (in bytes)    */
   ULONG  ulMemFlags  = 0;         /* Attribute flags for the object      */
   ULONG  ulMemSize   = 0;         /* Size of memory region for DosSetMem */
   APIRET rc          = NO_ERROR;  /* Return code                         */
```

```
   ulObjSize = 2000;          /* Will be rounded to a page boundary - 4096 */

   rc = DosAllocMem(&MyObject,     /* Pointer to memory object pointer     */
                    ulObjSize,     /* Size of object to be allocated       */
                    PAG_WRITE );   /* Allocate memory as read/writeable    */
   if (rc != NO_ERROR) {
      printf("DosAllocMem error:  return code = %u\n",rc);
      return 1;
   }
       /* Object can't be used until it is COMMITTED.  Since this was
          not done at DosAllocMem time, do it now.                    */

   rc = DosSetMem(MyObject,                    /* Pointer to object      */
                  ulObjSize,                   /* Size of area to change */
                  PAG_DEFAULT | PAG_COMMIT );  /* Commit the object      */

   if (rc != NO_ERROR) {
      printf("DosSetMem error:  return code = %u\n",rc);
      rc = DosFreeMem(MyObject); /* If omitted, OS/2 frees it at termination */
      return 1;
   } else { printf("DosSetMem: complete\n"); }

   strcpy(MyObject, "The memory object has just been used.");

              /* Check COMMIT status of the memory object. */

   ulMemSize = ulObjSize;
   rc = DosQueryMem(MyObject, &ulMemSize, &ulMemFlags);
   if (rc == NO_ERROR) {
      if (ulMemFlags & PAG_COMMIT) {
         printf("   Page containing MyObject is now committed.\n");
      } else {
         printf("Error:  Page containing MyObject has not been committed.\n");
      } /* endif */
   } else {
      printf("DosQueryMem error: return code = %u\n",rc);
   } /* endif */
   rc = DosFreeMem(MyObject);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error:  return code = %u\n",rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosFreeMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFreeModule

DosFreeModule - Syntax

Frees the reference to the dynamic link module for this process.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

HMODULE    hmod;  /*  The handle of the dynamic link module that is to be freed. */
APIRET     ulrc;  /*  Return Code. */

ulrc = DosFreeModule(hmod);
```

DosFreeModule Parameter - hmod

hmod (HMODULE) - input
The handle of the dynamic link module that is to be freed.

DosFreeModule Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFreeModule returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
12	ERROR_INVALID_ACCESS
95	ERROR_INTERRUPT

For a full list of error codes, see Errors.

DosFreeModule - Parameters

hmod (HMODULE) - input
The handle of the dynamic link module that is to be freed.

ulrc (APIRET) - returns
Return Code.

DosFreeModule returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
12	ERROR_INVALID_ACCESS
95	ERROR_INTERRUPT

For a full list of error codes, see Errors.

DosFreeModule - Remarks

DosFreeModule frees the reference to the dynamic link module for this process.

If the dynamic link module is no longer used by any process, the module is freed from system memory.

The module identified by the handle must have been loaded using DosLoadModule. If the handle is invalid, an error is returned.

After this function has completed, the module handle is no longer valid, and may not be used to refer to the dynamic link module. Procedure entry addresses returned for this module are also no longer valid, and will cause a protection fault if they are invoked.

DosFreeModule - Related Functions

Related Functions

- DosLoadModule
- DosQueryModuleName

DosFreeModule - Example Code

This example loads, queries, and then frees the dynamic link module "DISPLAY.DLL".

```
 #define INCL_DOSMODULEMGR     /* Module Manager values */
 #define INCL_DOSERRORS        /* Error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

 PSZ      ModuleName     = "C:\\OS2\\DLL\\DISPLAY.DLL";  /* Name of module   */
 UCHAR    LoadError[256] = "";          /* Area for Load failure information */
 HMODULE  ModuleHandle   = NULLHANDLE;  /* Module handle                     */
 PFN      ModuleAddr     = 0;           /* Pointer to a system function      */
 ULONG    ModuleType     = 0;           /* Module type                       */
 APIRET   rc             = NO_ERROR;    /* Return code                       */

   rc = DosLoadModule(LoadError,               /* Failure information buffer */
                      sizeof(LoadError),       /* Size of buffer             */
                      ModuleName,              /* Module to load             */
                      &ModuleHandle);          /* Module handle returned     */
   if (rc != NO_ERROR) {
      printf("DosLoadModule error: return code = %u\n", rc);
      return 1;
   } else {
      printf("Module %s loaded.\n", ModuleName);
   } /* endif */

   rc = DosQueryProcAddr(ModuleHandle,         /* Handle to module           */
                         1L,                   /* No ProcName specified      */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleAddr);         /* Address returned           */
   if (rc != NO_ERROR) {
      printf("DosQueryProcAddr error: return code = %u\n", rc);
      return 1;
   } else printf("Address of module is 0x%x.\n", ModuleAddr);

   rc = DosQueryProcType(ModuleHandle,         /* Handle to module           */
                         1L,                   /* Indicate no ProcName given */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleType);         /* Type 0=16-bit   1=32-bit   */
   if (rc != NO_ERROR) {
      printf("DosQueryProcType error: return code = %u\n", rc);
      return 1;
   } else printf("This is a %s module.\n", ( ModuleType ? "32-bit" : "16-bit"));

   rc = DosFreeModule(ModuleHandle);
   if (rc != NO_ERROR) {
      printf("DosFreeModule error: return code = %u\n", rc);
```

```
      return 1;
   } else printf("Module %s freed.\n", ModuleName);

   return NO_ERROR;
}
```

DosFreeModule - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFreeResource

DosFreeResource - Syntax

Frees a resource that was loaded by DosGetResource.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

PVOID     pb;    /*  The address of the resource to be freed. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosFreeResource(pb);
```

DosFreeResource Parameter - pb

pb (PVOID) - input
The address of the resource to be freed.

DosFreeResource Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFreeResource returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED

For a full list of error codes, see Errors.

DosFreeResource - Parameters

pb (PVOID) - input
The address of the resource to be freed.

ulrc (APIRET) - returns
Return Code.

DosFreeResource returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED

For a full list of error codes, see Errors.

DosFreeResource - Remarks

DosFreeResource frees a resource that was loaded by DosGetResource.

After the last reference to a resource is freed, the memory becomes available for reuse by the system; however, the memory is not reused until the system determines that it cannot satisfy a memory allocation request. This allows the resource to remain in memory in case the process issues DosGetResource again. The system thus avoids having to read the contents of the resource from the disk again.

DosFreeResource - Related Functions

Related Functions

- DosGetResource

DosFreeResource - Example Code

This example loads the dynamic link module "DISPLAY.DLL", gets a resource, and then releases it.

```
#define INCL_DOSRESOURCES     /* Resource types */
#define INCL_DOSMODULEMGR     /* Module Manager values */
#define INCL_DOSERRORS        /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR     LoadError[256] = "";         /* Area for Load failure information */
PSZ       ModuleName = "C:\\OS2\\DLL\\PMWP.DLL";  /* DLL with resources */
HMODULE   ModHandle  = NULLHANDLE;        /* Handle for module */
PVOID     Offset     = NULL;              /* Pointer to resource */
```

```
APIRET    rc         = NO_ERROR;          /* API return code */

 rc = DosLoadModule(LoadError,               /* Failure information buffer */
                    sizeof(LoadError),       /* Size of buffer             */
                    ModuleName,              /* Module to load             */
                    &ModHandle);             /* Module handle returned     */
 if (rc != NO_ERROR) {
    printf("DosLoadModule error: return code = %u\n", rc);
    return 1;
 }

 rc = DosGetResource(ModHandle,     /* Handle for DLL containing resources */
                     RT_POINTER,    /* Ask for  Pointer                    */
                     1L,            /*          with an ID of 1            */
                     &Offset);      /* Get back pointer                    */
 if (rc != NO_ERROR) {
    printf("DosGetResource error: return code = %u\n", rc);
    return 1;
 } else {
    printf("Resource Offset = 0x%x\n", Offset);
 } /* endif */

 rc = DosFreeResource(Offset);
 if (rc != NO_ERROR) {
    printf("DosFreeResource error: return code = %u\n", rc);
    return 1;
 }

return NO_ERROR;
}
```

DosFreeResource - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFreeThreadLocalMemory

DosFreeThreadLocalMemory - Syntax

Frees a block of thread-local memory that was originally allocated using DosAllocThreadLocalMemory.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     *p;   /*  Address of the memory block to free. */
APIRET     rc;  /*  Return Code. */

rc = DosFreeThreadLocalMemory(p);
```

DosFreeThreadLocalMemory Parameter - p

p (ULONG *) - input
Address of the memory block to free.

DosFreeThreadLocalMemory Return Value - rc

rc (APIRET) - returns
Return Code.

This function returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosFreeThreadLocalMemory - Parameters

p (ULONG *) - input
Address of the memory block to free.

rc (APIRET) - returns
Return Code.

This function returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosFreeThreadLocalMemory - Remarks

When a process is started, a small block of memory is set aside to be used as a thread-local memory area. This memory is at the same virtual address for each thread, but is backed by different physical memory. This permits each thread to have a small block of memory that is unique, or local, to that thread.

The thread-local memory area consists of 32 DWORDs (128 bytes), each DWORD being 32-bits in size.

DosFreeThreadLocalMemory - Related Functions

Related Functions

- DosAllocThreadLocalMemory

DosFreeThreadLocalMemory - Example Code

This example shows how to allocate and free 3 DWORDs of local thread memory.

```
#define  INCL_DOSPROCESS
#define  INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>          /* Needed for printf */

PULONG   pulThreadDWords = NULL;     /* Pointer to thread DWORDs returned */
APIRET   rc              = NO_ERROR; /* Return code */

int main(VOID) {

         /* Allocate 3 DWORDs of local thread memory */

  rc = DosAllocThreadLocalMemory(3,                    /* Number of DWORDs */
                                 &pulThreadDWords);    /* Address returned */

  if (rc != NO_ERROR) {
    printf("DosAllocThreadLocalMemory error: return code = %u\n", rc);
    return 1;
  }

          /* ... Use the thread-local memory ... */

  rc = DosFreeThreadLocalMemory(pulThreadDWords);     /* Free the DWORDs */

  if (rc != NO_ERROR) {
    printf("DosFreeThreadLocalMemory error: return code = %u\n", rc);
    return 1;
  }

  return NO_ERROR;
}
```

DosFreeThreadLocalMemory - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFSAttach

DosFSAttach - Syntax

Attaches or detaches a drive to or from a remote file system driver (FSD), or a pseudocharacter device name to or from a local or remote FSD.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszDevice;      /*  Pseudocharacter or spool device name. */
PSZ       pszFilesystem;  /*  Pointer to the remote file-system driver name. */
PVOID     pData;          /*  Pointer to file-system driver or spooler data. */
ULONG     cbData;         /*  The length, in bytes, of pData. */
ULONG     flag;           /*  Type of operation to be performed. */
APIRET    ulrc;           /*  Return Code. */

ulrc = DosFSAttach(pszDevice, pszFilesystem,
         pData, cbData, flag);
```

DosFSAttach Parameter - pszDevice

pszDevice (PSZ) - input
Pseudocharacter or spool device name.

A drive designation or a pseudocharacter device name when *flag* is 0 or 1. A drive designation is an ASCIIZ string consisting of the drive name followed by a colon. If an attachment is successful, all requests to that drive are routed to the specified file-system driver. If a detachment is successful, the drive is removed from the system's name space.

pszDevice points to the name of a spooled device when *flag* is 2 or 3. The *pszDevice* format is the same as above. Requests to that name are not seen by the file-system driver.

A pseudocharacter device name (single file device) is an ASCIIZ string consisting of the file-name subdirectory \DEV\. All requests to that name are routed to the specified file-system driver after a successful attachment. A successful detachment removes the name from the system's name space.

DosFSAttach Parameter - pszFilesystem

pszFilesystem (PSZ) - input
Pointer to the remote file-system driver name.

Address of the ASCIIZ name of the remote file-system driver that is to be attached to or detached from the device specified by *pszDevice* . For spooled objects, this pointer is set to 0. The pointer to *pszFilesystem* must be set to 0 when *flag* is 2 or 3.

DosFSAttach Parameter - pData

pData (PVOID) - input
Pointer to file-system driver or spooler data.

Address of the user-supplied file-system driver argument data area when *flag* is 0 or 1. The meaning of the data is specific to the file-system driver. *pData* contains contiguous ASCIIZ strings; the first word of the buffer contains the number of ASCIIZ strings. When *flag* is 2, *pData* points to the structure as follows:

hNmPipe (*USHORT*)
Handle of named pipe opened by spooler

cbSpoolObj (*BYTE*)
Length of name of spooler object (excluding NULL)

szSpoolObj (*UCHAR*)
Name of spooler object

When *flag* is 3, *pData* is set to zero.

DosFSAttach Parameter - cbData

cbData (ULONG) - input
The length, in bytes, of *pData*.

DosFSAttach Parameter - flag

flag (ULONG) - input
Type of operation to be performed.

Possible values shown in the following list:

0	FS_ATTACH Attach file server
1	FS_DETACH Detach file server
2	FS_SPOOLATTACH Register a spooler device
3	FS_SPOOLDETACH De-register a spooler device

DosFSAttach Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFSAttach returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
15	ERROR_INVALID_DRIVE
124	ERROR_INVALID_LEVEL
252	ERROR_INVALID_FSD_NAME
253	ERROR_INVALID_PATH

For a full list of error codes, see Errors.

DosFSAttach - Parameters

pszDevice (PSZ) - input
Pseudocharacter or spool device name.

A drive designation or a pseudocharacter device name when *flag* is 0 or 1. A drive designation is an ASCIIZ string consisting of the drive name followed by a colon. If an attachment is successful, all requests to that drive are routed to the specified file-system driver. If a detachment is successful, the drive is removed from the system's name space.

pszDevice points to the name of a spooled device when *flag* is 2 or 3. The *pszDevice* format is the same as above. Requests to that name are not seen by the file-system driver.

A pseudocharacter device name (single file device) is an ASCIIZ string consisting of the file-name subdirectory \DEV\. All requests to that name are routed to the specified file-system driver after a successful attachment. A successful detachment removes the name from the system's name space.

pszFilesystem (PSZ) - input
Pointer to the remote file-system driver name.

Address of the ASCIIZ name of the remote file-system driver that is to be attached to or detached from the device specified by *pszDevice* . For spooled objects, this pointer is set to 0. The pointer to *pszFilesystem* must be set to 0 when *flag* is 2 or 3.

pData (PVOID) - input
Pointer to file-system driver or spooler data.

Address of the user-supplied file-system driver argument data area when *flag* is 0 or 1. The meaning of the data is specific to the file-system driver. *pData* contains contiguous ASCIIZ strings; the first word of the buffer contains the number of ASCIIZ strings. When *flag* is 2, *pData* points to the structure as follows:

hNmPipe (*USHORT*)
Handle of named pipe opened by spooler

cbSpoolObj (*BYTE*)
Length of name of spooler object (excluding NULL)

szSpoolObj (*UCHAR*)
Name of spooler object

When *flag* is 3, *pData* is set to zero.

cbData (ULONG) - input
The length, in bytes, of *pData* .

flag (ULONG) - input
Type of operation to be performed.

Possible values shown in the following list:

0	FS_ATTACH Attach file server
1	FS_DETACH Detach file server
2	FS_SPOOLATTACH Register a spooler device
3	FS_SPOOLDETACH De-register a spooler device

ulrc (APIRET) - returns
Return Code.

DosFSAttach returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
15	ERROR_INVALID_DRIVE
124	ERROR_INVALID_LEVEL
252	ERROR_INVALID_FSD_NAME
253	ERROR_INVALID_PATH

For a full list of error codes, see Errors.

DosFSAttach - Remarks

The redirection of drive letters that represent local drives is not supported.

File-system drivers cannot use DosFSAttach to establish open connections that are not attached to a name in the system's name space. They must issue DosFSCtl for such purposes as optimizing UNC connections or establishing access rights. DosFSAttach creates attachments only to drives or devices in the system's name space.

DosFSAttach - Related Functions

Related Functions

- DosFSCtl

DosFSAttach - Example Code

This example attaches a drive to a remote file system driver (FSD). Assume that the FSD does not require any user-supplied data arguments.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

 PVOID   pvDataBuffer    = NULL;     /* Arguments for the file system driver */
 APIRET  rc              = NO_ERROR; /* Return code */

    rc = DosFSAttach("Q:",                  /* Drive letter */
                     "\\LAN\\LANTOOLS",     /* Remote file system driver */
                     pvDataBuffer,          /* User-supplied arguments */
                     0,                     /* No arguments supplied */
                     FS_ATTACH);            /* Attach to the file system */

    if (rc != NO_ERROR) {
        printf("DosFSAttach error: return code = %u\n", rc);
        return 1;
    }

    return NO_ERROR;
}
```

DosFSAttach - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosFSCtl

DosFSCtl - Syntax

Provides an extended standard interface between an application and a file-system driver (FSD).

```
#define INCL_DOSFILEMGR
#include <os2.h>

PVOID     pData;     /*  Address of the data area. */
ULONG     cbData;    /*  The length, in bytes, of pData. */
PULONG    pcbData;   /*  Pointer to the length of data passed to or returned from the FSD. */
PVOID     pParms;    /*  Address of the command-specific parameter list. */
ULONG     cbParms;   /*  The length, in bytes, of pParms. */
PULONG    pcbParms;  /*  Pointer to the length of the parameters passed to or returned from the FSD. */
ULONG     function;  /*  The function code that is specific to the file-system driver. */
PSZ       pszRoute;  /*  Address of the ASCIIZ name of the FSD, or the path name of a file or directory that the
HFILE     hFile;     /*  File-specific or device-specific handle. */
ULONG     method;    /*  Method used to routed the request. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosFSCtl(pData, cbData, pcbData, pParms,
         cbParms, pcbParms, function, pszRoute,
         hFile, method);
```

DosFSCtl Parameter - pData

pData (PVOID) - input
Address of the data area.

DosFSCtl Parameter - cbData

cbData (ULONG) - input
The length, in bytes, of *pData* .

This is the maximum length of the data to be returned by the file-system driver in *pData* . *pcbData* may be larger than this on input, but not on output.

DosFSCtl Parameter - pcbData

pcbData (PULONG) - in/out

Pointer to the length of data passed to or returned from the FSD.

Input	Pointer to the length, in bytes, of the data passed to the file-system driver in *pData*
Output	Pointer to the length, in bytes, of the data returned by the file-system driver in *pData* . If this function returns ERROR_BUFFER_OVERFLOW, *pcbData* points to the size of the buffer required to hold the data returned by the file-system driver.

DosFSCtl Parameter - pParms

pParms (PVOID) - input
Address of the command-specific parameter list.

DosFSCtl Parameter - cbParms

cbParms (ULONG) - input
The length, in bytes, of *pParms* .

This is the maximum length of the data to be returned by the file-system driver in *pParms* . *pcbParms* may be larger than this on input, but not on output.

DosFSCtl Parameter - pcbParms

pcbParms (PULONG) - in/out
Pointer to the length of the parameters passed to or returned from the FSD.

Input	Pointer to the length, in bytes, of the parameters passed to the file-system driver in *pParms* .
Output	Pointer to the length, in bytes, of the parameters returned by the file-system driver in *pParms* . If this function returns ERROR_BUFFER_OVERFLOW, *pcbParms* . points to the size of the buffer required to hold the parameters returned by the file-system driver. No other data is returned in this case.

DosFSCtl Parameter - function

function (ULONG) - input
The function code that is specific to the file-system driver.

For remote file-system drivers, two kinds of DosFSCtl functions are possible: functions that are handled locally, and functions that are exported across the network. If bit 0x4000 is set in *function* , this indicates to the remote file-system driver (FSD) that the function should be exported.

Function codes from 0x0000 to 0x7FFF are reserved for use by the operating system. Function codes from 0x8000 to 0xBFFF are FSD-defined DosFSCtl functions handled by the local file-system driver. Function codes from 0xC000 to 0xFFFF are FSD-defined DosFSCtl functions exported to the server.

function may have one of the following values:

1 FSCTL_ERROR_INFO
Returns error-code information from the file-system driver.

Input — The error code is passed to the file-system driver in the first word of *pParms* .

Output — The ASCIIZ string returned in *pData* is an explanation of the error code.

2 FSCTL_MAX_EASIZE
Queries the file-system driver for the maximum size of individual EAs (extended attributes), and the maximum size of the full EA list that it supports. The information is returned in *pData* in the form of an EASIZEBUF structure.

DosFSCtl Parameter - pszRoute

pszRoute (PSZ) - input
Address of the ASCIIZ name of the FSD, or the path name of a file or directory that the operation applies to.

This parameter must be a null pointer (OL) If *method* is equal to FSCTL_HANDLE

DosFSCtl Parameter - hFile

hFile (HFILE) - input
File-specific or device-specific handle.

This parameter must be -1 when *method* is equal to FSCTL_PATHNAME or FSCTL_FSDNAME.

DosFSCtl Parameter - method

method (ULONG) - input
Method used to routed the request.

Possible values are shown in the following list:

1 FSCTL_HANDLE
hFile directs routing. *pszRoute* must be a null pointer (0L). The file-system driver associated with the handle receives the request.

2 FSCTL_PATHNAME
pszRoute refers to a path name that directs routing. *hFile* must be -1. The file-system driver associated with the drive that the path name refers to at the time of the request receives the request. The path name need not refer to a file or directory that actually exists, only to a drive. A relative path name may be used; it is processed like any other path name.

3 FSCTL_FSDNAME
pszRoute refers to a file-system driver name that directs routing. *hFile* must be -1. The named file-system driver receives the request.

DosFSCtl Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosFSCtl returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
111	ERROR_BUFFER_OVERFLOW
117	ERROR_INVALID_CATEGORY
124	ERROR_INVALID_LEVEL
252	ERROR_INVALID_FSD_NAME

For a full list of error codes, see Errors.

DosFSCtl - Parameters

pData (PVOID) - input
Address of the data area.

cbData (ULONG) - input
The length, in bytes, of *pData* .

This is the maximum length of the data to be returned by the file-system driver in *pData* . *pcbData* may be larger than this on input, but not on output.

pcbData (PULONG) - in/out
Pointer to the length of data passed to or returned from the FSD.

Input	Pointer to the length, in bytes, of the data passed to the file-system driver in *pData*
Output	Pointer to the length, in bytes, of the data returned by the file-system driver in *pData* . If this function returns ERROR_BUFFER_OVERFLOW, *pcbData* points to the size of the buffer required to hold the data returned by the file-system driver.

pParms (PVOID) - input
Address of the command-specific parameter list.

cbParms (ULONG) - input
The length, in bytes, of *pParms* .

This is the maximum length of the data to be returned by the file-system driver in *pParms* . *pcbParms* may be larger than this on input, but not on output.

pcbParms (PULONG) - in/out
Pointer to the length of the parameters passed to or returned from the FSD.

Input	Pointer to the length, in bytes, of the parameters passed to the file-system driver in *pParms* .
Output	Pointer to the length, in bytes, of the parameters returned by the file-system driver in *pParms* . If this function returns ERROR_BUFFER_OVERFLOW, *pcbParms* . points to the size of the buffer required to hold the parameters returned by the file-system driver. No other data is returned in this case.

function (ULONG) - input
The function code that is specific to the file-system driver.

For remote file-system drivers, two kinds of DosFSCtl functions are possible: functions that are handled locally, and functions that are exported across the network. If bit 0x4000 is set in *function* , this indicates to the remote file-system driver (FSD) that the function should be exported.

Function codes from 0x0000 to 0x7FFF are reserved for use by the operating system. Function codes from 0x8000 to 0xBFFF are FSD-defined DosFSCtl functions handled by the local file-system driver. Function codes from 0xC000 to 0xFFFF are FSD-defined

DosFSCtl functions exported to the server.

function may have one of the following values:

1 FSCTL_ERROR_INFO
Returns error-code information from the file-system driver.

Input	The error code is passed to the file-system driver in the first word of *pParms* .
Output	The ASCIIZ string returned in *pData* is an explanation of the error code.

2 FSCTL_MAX_EASIZE
Queries the file-system driver for the maximum size of individual EAs (extended attributes), and the maximum size of the full EA list that it supports. The information is returned in *pData* in the form of an EASIZEBUF structure.

pszRoute (PSZ) - input
Address of the ASCIIZ name of the FSD, or the path name of a file or directory that the operation applies to.

This parameter must be a null pointer (OL) If *method* is equal to FSCTL_HANDLE

hFile (HFILE) - input
File-specific or device-specific handle.

This parameter must be -1 when *method* is equal to FSCTL_PATHNAME or FSCTL_FSDNAME.

method (ULONG) - input
Method used to routed the request.

Possible values are shown in the following list:

1 FSCTL_HANDLE
hFile directs routing. *pszRoute* must be a null pointer (0L). The file-system driver associated with the handle receives the request.

2 FSCTL_PATHNAME
pszRoute refers to a path name that directs routing. *hFile* must be -1. The file-system driver associated with the drive that the path name refers to at the time of the request receives the request. The path name need not refer to a file or directory that actually exists, only to a drive. A relative path name may be used; it is processed like any other path name.

3 FSCTL_FSDNAME
pszRoute refers to a file-system driver name that directs routing. *hFile* must be -1. The named file-system driver receives the request.

ulrc (APIRET) - returns
Return Code.

DosFSCtl returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
111	ERROR_BUFFER_OVERFLOW
117	ERROR_INVALID_CATEGORY
124	ERROR_INVALID_LEVEL
252	ERROR_INVALID_FSD_NAME

For a full list of error codes, see Errors.

DosFSCtl - Remarks

For debugging considerations, see DosDebug.

DosFSCtl - Related Functions

Related Functions

- DosFSAttach

DosFSCtl - Example Code

The following is NOT a complete C program. It is intended to provide an idea of how to communicate with a file system driver (FSD).

This example assumes that FileHandle has been initialized with the handle to the file and that the file system driver (FSD) recognizes a function code of hex 81DE. It further assumes that the input parameters and input data area are appropriate for the function.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

UCHAR   uchDataArea[200] = {0};          /* Input and output data area */
ULONG   ulDataLen        = 0;            /* Input and output data size */
UCHAR   uchParms[120]    = {0};          /* Input and output for function */
ULONG   ulParmLen        = 0;            /* Input and output parameter size */
ULONG   ulFunction       = 0x81DE;       /* Device-specific function */
HFILE   hfFile           = NULLHANDLE;   /* Handle for file */
APIRET  rc               = NO_ERROR;     /* Return code */

 strcpy(uchDataArea,"34 22 37");   /* Data to pass to file system */
 ulDataLen = strlen(uchDataArea);  /* Length of input data */

 strcpy(uchParms,"PARM1: 98");     /* Input parameters */
 ulParmLen = strlen(uchParms);     /* Length of input parameters */

 rc = DosFSCtl(uchDataArea,          /* Input/output data area */
               sizeof(uchDataArea), /* Maximum output data size */
               &ulDataLen,          /* Input:  size of input data area */
                                    /* Output: size of data returned   */
               uchParms,            /* Input/Output parameter list */
               sizeof(uchParms),    /* Maximum output parameter size */
               &ulParmLen,          /* Input:  size of parameter list */
                                    /* Output: size of parameters returned */
               ulFunction,          /* Function being requested */
               "MY_FSD",            /* File System Driver (FSD) name */
               hfFile,              /* Handle for file */
               FSCTL_FSDNAME);      /* Indicate FSD name is the route */

 if (rc != NO_ERROR) {
     printf("DosFSCtl error: return code = %u\n", rc);
     return 1;
 }
```

DosFSCtl - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGetDateTime

DosGetDateTime - Syntax

Gets the current date and time.

```
#define INCL_DOSDATETIME
#include <os2.h>

PDATETIME    pdt;   /*  Pointer to the DateTime data structure. */
APIRET       ulrc;  /*  Return Code. */

ulrc = DosGetDateTime(pdt);
```

DosGetDateTime Parameter - pdt

pdt (PDATETIME) - output
Pointer to the DateTime data structure.

DosGetDateTime Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGetDateTime returns no values.

DosGetDateTime - Parameters

pdt (PDATETIME) - output
Pointer to the DateTime data structure.

ulrc (APIRET) - returns
Return Code.

DosGetDateTime returns no values.

DosGetDateTime - Remarks

DosGetDateTime gets the date and time that are maintained by the operating system.

To set the date and time, issue DosSetDateTime.

DosGetDateTime - Related Functions

Related Functions

- DosAsyncTimer
- DosSetDateTime
- DosSleep
- DosStartTimer
- DosStopTimer

DosGetDateTime - Example Code

This example displays the current date in country-dependent format.

```
#define INCL_DOSNLS        /* National Language Support values */
#define INCL_DOSDATETIME   /* Date and time values */
#define INCL_DOSERRORS     /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

COUNTRYCODE  Country    = {0};   /* Country code info (0 = current country) */
COUNTRYINFO  CtryInfo   = {0};   /* Buffer for country-specific information */
ULONG        ulInfoLen  = 0;
DATETIME     DateTime   = {0};       /* Date and time information           */
APIRET       rc         = NO_ERROR;  /* Return code                         */

  rc = DosQueryCtryInfo(sizeof(CtryInfo), &Country,
                        &CtryInfo, &ulInfoLen);

  if (rc != NO_ERROR) {
      printf("DosQueryCtryInfo error: return code = %u\n",rc);
      return 1;
  }

  rc = DosGetDateTime(&DateTime);  /* Retrieve the current date and time  */

  if (rc != NO_ERROR) {
     printf ("DosGetDateTime error : return code = %u\n", rc);
     return 1;
  } else {
     switch (CtryInfo.fsDateFmt) {

     case(1):                                              /* dd/mm/yy */
     printf("Today is %d%s%d%s%d\n", DateTime.day, CtryInfo.szDateSeparator,
             DateTime.month, CtryInfo.szDateSeparator, DateTime.year);
       break;

     case(2):                                              /* yy/mm/dd */
     printf("Today is %d%s%d%s%d\n", DateTime.year, CtryInfo.szDateSeparator,
             DateTime.month, CtryInfo.szDateSeparator, DateTime.day);
       break;

     default:                                              /* mm/dd/yy */
     printf("Today is %d%s%d%s%d\n", DateTime.month, CtryInfo.szDateSeparator,
             DateTime.day, CtryInfo.szDateSeparator, DateTime.year);
       break;

      } /* endswitch */
  }
    return NO_ERROR;
  }
```

DosGetDateTime - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGetInfoBlocks

DosGetInfoBlocks - Syntax

Gets the address of:

- The thread information block (TIB) of the current thread
- The process information block (PIB) of the current process

```
#define INCL_DOSPROCESS
#include <os2.h>

PTIB     *pptib;  /*  A pointer to the address of the TIB in which the current thread is returned. */
PPIB     *pppib;  /*  The address of a pointer to the PIB in which the current process is returned. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosGetInfoBlocks(pptib, pppib);
```

DosGetInfoBlocks Parameter - pptib

pptib (PTIB *) - output
A pointer to the address of the TIB in which the current thread is returned.

You can also specify NULL in this field if you do not want the current address of the TIB returned. See **Remarks** for a description of the TIB.

DosGetInfoBlocks Parameter - pppib

pppib (PPIB *) - output

The address of a pointer to the PIB in which the current process is returned.

You can also specify NULL in this field if you do not want the current address of the PIB returned. See **Remarks** for a description of the PIB.

DosGetInfoBlocks Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGetInfoBlocks returns no values.

DosGetInfoBlocks - Parameters

pptib (PTIB *) - output
A pointer to the address of the TIB in which the current thread is returned.

You can also specify NULL in this field if you do not want the current address of the TIB returned. See **Remarks** for a description of the TIB.

pppib (PPIB *) - output
The address of a pointer to the PIB in which the current process is returned.

You can also specify NULL in this field if you do not want the current address of the PIB returned. See **Remarks** for a description of the PIB.

ulrc (APIRET) - returns
Return Code.

DosGetInfoBlocks returns no values.

DosGetInfoBlocks - Remarks

DosGetInfoBlocks returns the address of the TIB of the current thread. This function also returns the address of the PIB of the current process.

Several items of per-thread information are kept in a read/write area of the process address space called the Thread Information Block, or TIB. You can access this information by calling DosGetInfoBlocks.

TIB2 contains system-specific thread information.

Several items of per-process information are kept in a read/write area of the process address space called the Process Information Block, or PIB. You can access this information by calling DosGetInfoBlocks.

DosGetInfoBlocks - Related Functions

Related Functions

- DosCreateThread

DosGetInfoBlocks - Example Code

This sample sets the current thread priority to Time Critical level 15. It then uses DosGetInfoBlocks to retrieve the priority.

```
#define INCL_DOSPROCESS
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

int main(VOID)
{
   PTIB   ptib = NULL;          /* Thread information block structure  */
   PPIB   ppib = NULL;          /* Process information block structure */
   APIRET rc   = NO_ERROR;      /* Return code                         */

   rc = DosSetPriority (PRTYS_THREAD,        /* Change a single thread */
                        PRTYC_TIMECRITICAL,  /* Time critical class    */
                        15L,                 /* Increase by 15         */
                        0L);                 /* Assume current thread  */
   if (rc != NO_ERROR) {
      printf ("DosSetPriority error : rc = %u\n", rc);
      return 1;
   } else {
      rc = DosGetInfoBlocks(&ptib, &ppib);

      if (rc != NO_ERROR) {
         printf ("DosGetInfoBlocks error : rc = %u\n", rc);
         return 1;
      } else {
         printf("Priority Class = %d\n",
                ((ptib->tib_ptib2->tib2_ulpri) >> 8) & 0x00FF);
         printf("Priority Level = %d\n",
                ((ptib->tib_ptib2->tib2_ulpri) & 0x001F) );
      } /* endif */
   }

   return NO_ERROR;
}
```

If you only want the address of the TIB returned, code the DosGetInfoBlocks call as follows:

```
      rc = DosGetInfoBlocks(&ptib, NULL);
```

If you only want the address of the PIB returned, code the DosGetInfoBlocks call as follows:

```
      rc = DosGetInfoBlocks(NULL, &ppib.);
```

DosGetInfoBlocks - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGetMessage

DosGetMessage - Syntax

Retrieves a message from the specified system message file, and inserts variable text-string information into the message.

```
#define INCL_DOSMISC
#include <os2.h>

PCHAR     *pTable;     /*  Pointer table. */
ULONG      cTable;     /*  Number of variable insertion text strings. */
PCHAR      pBuf;       /*  The address of the caller's buffer area where the system returns the requested messag
ULONG      cbBuf;      /*  The length, in bytes, of the caller's buffer area. */
ULONG      msgnumber;  /*  The message number requested. */
PSZ        pszFile;    /*  The drive designation, path, and name of the file where the message can be found. */
PULONG     pcbMsg;     /*  The actual length, in bytes, of the message returned. */
APIRET     ulrc;       /*  Return Code. */

ulrc = DosGetMessage(pTable, cTable, pBuf,
         cbBuf, msgnumber, pszFile, pcbMsg);
```

DosGetMessage Parameter - pTable

pTable (PCHAR *) - input
Pointer table.

Each doubleword pointer points to an ASCIIZ string or a double-byte character-set (DBCS) string ending in nulls. A maximum of nine strings can be present.

DosGetMessage Parameter - cTable

cTable (ULONG) - input
Number of variable insertion text strings.

Possible values range from 0 to 9. If *cTable* is 0, *pTable* is ignored.

DosGetMessage Parameter - pBuf

pBuf (PCHAR) - output
The address of the caller's buffer area where the system returns the requested message.

If the message is too long to fit in the caller's buffer, then as much of the message text as possible is returned, with the appropriate

error return code.

DosGetMessage Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of the caller's buffer area.

DosGetMessage Parameter - msgnumber

msgnumber (ULONG) - input
The message number requested.

DosGetMessage Parameter - pszFile

pszFile (PSZ) - input
The drive designation, path, and name of the file where the message can be found.

The drive designation and path are optional. This specifies a file that was previously prepared by the MKMSGF utility program.

DosGetMessage Parameter - pcbMsg

pcbMsg (PULONG) - output
The actual length, in bytes, of the message returned.

DosGetMessage Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGetMessage returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
37	ERROR_CODE_PAGE_MISMATCHED
87	ERROR_INVALID_PARAMETER
206	ERROR_FILENAME_EXCED_RANGE
316	ERROR_MR_MSG_TOO_LONG
317	ERROR_MR_MID_NOT_FOUND
318	ERROR_MR_UN_ACC_MSGF
319	ERROR_MR_INV_MSGF_FORMAT

320	ERROR_MR_INV_IVCOUNT
321	ERROR_MR_UN_PERFORM

For a full list of error codes, see Errors.

DosGetMessage - Parameters

pTable (PCHAR *) - input
Pointer table.

Each doubleword pointer points to an ASCIIZ string or a double-byte character-set (DBCS) string ending in nulls. A maximum of nine strings can be present.

cTable (ULONG) - input
Number of variable insertion text strings.

Possible values range from 0 to 9. If *cTable* is 0, *pTable* is ignored.

pBuf (PCHAR) - output
The address of the caller's buffer area where the system returns the requested message.

If the message is too long to fit in the caller's buffer, then as much of the message text as possible is returned, with the appropriate error return code.

cbBuf (ULONG) - input
The length, in bytes, of the caller's buffer area.

msgnumber (ULONG) - input
The message number requested.

pszFile (PSZ) - input
The drive designation, path, and name of the file where the message can be found.

The drive designation and path are optional. This specifies a file that was previously prepared by the MKMSGF utility program.

pcbMsg (PULONG) - output
The actual length, in bytes, of the message returned.

ulrc (APIRET) - returns
Return Code.

DosGetMessage returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
37	ERROR_CODE_PAGE_MISMATCHED
87	ERROR_INVALID_PARAMETER
206	ERROR_FILENAME_EXCED_RANGE
316	ERROR_MR_MSG_TOO_LONG
317	ERROR_MR_MID_NOT_FOUND
318	ERROR_MR_UN_ACC_MSGF
319	ERROR_MR_INV_MSGF_FORMAT
320	ERROR_MR_INV_IVCOUNT
321	ERROR_MR_UN_PERFORM

For a full list of error codes, see Errors.

DosGetMessage - Remarks

DosGetMessage retrieves a message from the specified system message file, and inserts variable text-string information into the message.

If *cTable* is greater than 9, DosGetMessage returns an error indicating that *cTable* is out of range.

If the numeric value of x in the %x sequence for %1 to %9 is less than or equal to *cTable* , then text insertion, through substitution for %x, is performed for all occurrences of %x in the message. Otherwise, text insertion is ignored, and the %x sequence is returned in the message unchanged. Text insertion is performed for all text strings defined by *cTable* and *pTable* .

Variable data insertion does not depend on blank character delimiters, nor are blanks automatically inserted.

For warning and error messages, the 7-character message ID (3-character component ID concatenated with a 4-digit message number) followed by a colon and a blank character is returned as part of the message text. DosGetMessage determines the type of message based on the message classification generated in the output file of MKMSGF.

The following is an example of a sample error message returned with the message ID:

```
SYS0002: The system cannot find the file specified
```

DosGetMessage retrieves messages previously prepared by MKMSGF to create a message file, or by MSGBIND to bind a message segment to an .EXE file. First, DosGetMessage tries to retrieve the message from memory in the message segment bound to the .EXE program. If the message cannot be found, DosGetMessage retrieves the message from the message file on DASD (direct access storage device, such as a diskette or fixed-disk).

If the file name is not fully qualified, DosGetMessage searches the following directories for the default drive and path:

1. The system root directory
2. The current working directory
3. Directories listed in the DPATH (protect-mode) statement
4. Directories listed in the APPEND (DOS session) statement.

If a message cannot be retrieved because of a DASD error or a file-not-found condition, the system places an error message into the user's buffer area.

The following error conditions cause the system to place an error message into the user's buffer area:

- Unable to format the system message

 An error message is returned as a result of an invalid parameter (for example, invalid message number or invalid *cTable*).

- Unable to read the system message file

 An error message is returned when the system message file cannot be read (for example, because of a DASD error or an invalid message-file format).

- Unable to find the system message file

 An error message is returned when the system message file cannot be found.

The presence of the message in memory (EXE bound) or on DASD is not apparent to the caller, and is handled by DosGetMessage. In both cases, you refer to the message by message number and file name.

For DosGetMessage to be called from an input/output privilege level (IOPL) code segment, the following statement must be in the program's definition (.DEF) file:

```
SEGMENT '_MSGSEG' CLASS 'MSGSEGCODE' IOPL CONFORMING
```

In OS/2 Version 2.00, the message segment or object is packed with other application code. If the size of the code segment or object and the bound messages exceeds 64KB, then the message segment or object may be isolated from the application program code by placing the following statement into the program's definition (.DEF) file:

```
SEGMENT  '_MSGSEG' CLASS 'CODE' LOADONCALL    (16-bit application)

SEGMENT  '_MSGSEG32' CLASS 'CODE' LOADONCALL  (32-bit application)
```

DosGetMessage - Related Functions

Related Functions

- DosInsertMessage
- DosPutMessage
- DosQueryMessageCP

DosGetMessage - Example Code

This example demonstrates how to create a message with inserts from a system message file. It creates a file named "SAMPLE.TXT" with the following 3 lines in it: compact.

sample system message file - SAMPLE.MSG ...

DOS

DOS1000E: %1 Error at Station %2%0

It then generates a message file by issuing the following command at an OS/2 command prompt:

MKMSGF SAMPLE.TXT SAMPLE.MSG

```
#define INCL_DOSMISC    /* Miscellaneous values */
#define INCL_DOSERRORS  /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
UCHAR   *IvTable[2] = {0};                  /* Table of variables to insert */
UCHAR   szOutMsg[80]= "";                   /* Message buffer */
ULONG   ulMsgLen    = 0;                    /* Length of returned message */
APIRET  rc          = 0;                    /* Return code */

IvTable[0] = "Automation Failure";
IvTable[1] = "69B";

    /* Create error message with inserts from system message file SAMPLE.MSG */

rc = DosGetMessage(IvTable,          /* Message insert pointer array */
                   2,                /* Number of inserts */
                   szOutMsg,         /* Output message */
                   sizeof(szOutMsg), /* Length of output message area */
                   1000L,            /* Number of message requested */
                   "SAMPLE.MSG",     /* Message file (created by MKMSGF) */
                   &ulMsgLen);       /* Length of resulting output message */
if (rc != NO_ERROR) {
   printf("DosGetMessage error: return code = %u\n", rc);
   return 1;
}

printf("%s\n", szOutMsg);

return NO_ERROR;
}
```

DosGetMessage - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGetNamedSharedMem

DosGetNamedSharedMem - Syntax

Obtains access to a named shared memory object.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PPVOID    ppb;      /*  A pointer to a variable that will receive the base address of the shared memory object.
PSZ       pszName;  /*  The address of the name string associated with the shared memory object. */
ULONG     flag;     /*  A set of attribute flags that specify the desired access protection for the shared memor
APIRET    ulrc;     /*  Return Code. */

ulrc = DosGetNamedSharedMem(ppb, pszName,
         flag);
```

DosGetNamedSharedMem Parameter - ppb

ppb (PPVOID) - output
A pointer to a variable that will receive the base address of the shared memory object.

DosGetNamedSharedMem Parameter - pszName

pszName (PSZ) - input
The address of the name string associated with the shared memory object.

The name is an ASCIIZ string in the format of an OS/2 file name, and is in the subdirectory \SHARMEM\, for example, \SHAREMEM\PUBLIC.DAT.

DosGetNamedSharedMem Parameter - flag

flag (ULONG) - input
A set of attribute flags that specify the desired access protection for the shared memory object.

Desired Access Protection

The following access protections are available:

1 PAG_READ
Read access

2 PAG_WRITE
Write access

4 PAG_EXECUTE
Execute access to the committed pages in the shared memory object.

8 PAG_GUARD
Access to the committed pages in the shared memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

DosGetNamedSharedMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGetNamedSharedMem returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
8	ERROR_NOT_ENOUGH_MEMORY
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
123	ERROR_INVALID_NAME
212	ERROR_LOCKED

For a full list of error codes, see Errors.

DosGetNamedSharedMem - Parameters

ppb (PPVOID) - output
A pointer to a variable that will receive the base address of the shared memory object.

pszName (PSZ) - input
The address of the name string associated with the shared memory object.

The name is an ASCIIZ string in the format of an OS/2 file name, and is in the subdirectory \SHARMEM\, for example, \SHAREMEM\PUBLIC.DAT.

flag (ULONG) - input
A set of attribute flags that specify the desired access protection for the shared memory object.

Desired Access Protection

The following access protections are available:

1 PAG_READ
Read access

2 PAG_WRITE
Write access

4 PAG_EXECUTE
Execute access to the committed pages in the shared memory object.

8 PAG_GUARD
Access to the committed pages in the shared memory object causes a "guard page entered" condition to be raised

in the subject process.

At least one of the bits PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

ulrc (APIRET) - returns
Return Code.

DosGetNamedSharedMem returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
8	ERROR_NOT_ENOUGH_MEMORY
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
123	ERROR_INVALID_NAME
212	ERROR_LOCKED

For a full list of error codes, see Errors.

DosGetNamedSharedMem - Remarks

DosGetNamedSharedMem obtains access to a named shared memory object.

Getting a named shared memory object allocates the virtual address (of the shared memory object) in the virtual-address space of the process.

When the name of the shared memory object is specified, the name string provided must include the prefix "\SHAREMEM\".

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

The value *ppb* returned to the process issuing this function is the same as that returned to the process that created the shared memory object.

Error code 31 (ERROR_GEN_FAILURE) is returned when the segment's maximum reference count of 65535 is exceeded.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosGetNamedSharedMem - Related Functions

Related Functions

- DosAllocSharedMem
- DosGetSharedMem
- DosGiveSharedMem

DosGetNamedSharedMem - Example Code

This example allocates named shared memory, and shows how another process can access it. It finally frees the memory object.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main (VOID)
{
   PVOID  pvShrObject = NULL;      /* Pointer to shared memory object      */
   PSZ    pszMemName  = "\\SHAREMEM\\MYTOOL\\APPLICAT.DAT"; /* Object name */
   PVOID  pvAltObject = NULL;      /* Alternate pointer to shared memory   */
   APIRET rc          = NO_ERROR;  /* Return code                          */
   ULONG  ulObjSize   = 1024;   /* Size (system rounds to 4096 - page bdy) */

   rc = DosAllocSharedMem(&pvShrObject,   /* Pointer to object pointer     */
                          pszMemName,     /* Name for shared memory        */
                          ulObjSize,      /* Desired size of object        */
                          PAG_COMMIT |    /* Commit memory                 */
                          PAG_WRITE );    /* Allocate memory as read/write */
   if (rc != NO_ERROR) {
      printf("DosAllocSharedMem error:  return code = %u\n",rc);
      return 1;
   }

   strcpy(pvShrObject, "Write your shared application data here.");

      /* Get the address of the shared memory and reference it that way.
         (Done for illustrative purposes only, this is how another process
         would go about accessing the named shared memory.)            */

   rc = DosGetNamedSharedMem(&pvAltObject,  /* Pointer to pointer of object */
                             pszMemName,    /* Name of shared memory        */
                             PAG_READ);     /* Want read-only access        */
   if (rc != NO_ERROR) {
      printf("DosGetNamedSharedMem error:  return code = %u\n",rc);
      return 1;
   }

   printf("Shared data read was \"%s\"\n",pvAltObject);

   rc = DosFreeMem(pvShrObject);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error:  return code = %u\n",rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosGetNamedSharedMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGetResource

DosGetResource - Syntax

Returns a pointer to a specified resource.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

HMODULE    hmod;    /*  Handle of module that has the required resource. */
ULONG      idType;  /*  The type identifier of the 32-bit resource. */
ULONG      idName;  /*  The name identifier of the 32-bit resource. */
PPVOID     ppb;     /*  A pointer to the resource. */
APIRET     ulrc;    /*  Return Code. */

ulrc = DosGetResource(hmod, idType, idName,
         ppb);
```

DosGetResource Parameter - hmod

hmod (HMODULE) - input
Handle of module that has the required resource.

A value of zero means to get the address from the current process. A value other than zero is a module handle that was returned by DosLoadModule.

DosGetResource Parameter - idType

idType (ULONG) - input
The type identifier of the 32-bit resource.

This value must be in the range of 0x0000 to 0xFFFF.

DosGetResource Parameter - idName

idName (ULONG) - input
The name identifier of the 32-bit resource.

This value must be in the range of 0x0000 to 0xFFFF.

DosGetResource Parameter - ppb

ppb (PPVOID) - output
A pointer to the resource.

DosGetResource Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGetResource returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosGetResource - Parameters

hmod (HMODULE) - input
Handle of module that has the required resource.

A value of zero means to get the address from the current process. A value other than zero is a module handle that was returned by DosLoadModule.

idType (ULONG) - input
The type identifier of the 32-bit resource.

This value must be in the range of 0x0000 to 0xFFFF.

idName (ULONG) - input
The name identifier of the 32-bit resource.

This value must be in the range of 0x0000 to 0xFFFF.

ppb (PPVOID) - output
A pointer to the resource.

ulrc (APIRET) - returns
Return Code.

DosGetResource returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosGetResource - Remarks

DosGetResource returns the address of the specified resource object.

Resource objects are read-only data objects that can be accessed dynamically at run time. The access key is two 32-bit numbers. The first

number is a type ID; the second, a name ID. These are similar to the file extension and file-name portions of a file name.

Resource objects are placed into an executable file by the Resource Compiler (RC.EXE).

If a STRINGTABLE or MESSAGETABLE resource is specified in an RC file, with a number of IDs and respective strings, these RT_STRING or RT_MESSAGE string resources are bundled together in groups of 16. Any missing IDs are replaced with zero length strings. Each group, or bundle, of RT_STRING or RT_MESSAGE resources is assigned a unique sequential ID. The resource string ID is not necessarily the same as the ID specified when using DosGetResource. The formula for calculating the ID of the resource bundle, for use in DosGetResource, is as follows:

```
bundle ID = ( id / 16) +1
```

where id is the string ID assigned in the RC file.

Thus, bundle 1 contains strings 0 to 15, bundle 2 contains strings 16 to 31, and so on. Once the address of the bundle has been returned by DosGetResource (using the calculated ID), the buffer can be parsed to locate the particular string within the bundle. The number of the string is calculated by the formula:

```
string = id % 16
```

(string = remainder for id/16).

The buffer returned consists of the CodePage of the strings in the first USHORT, followed by the 16 strings in the bundle. The first BYTE of each string is the length of the string (including the null terminator), followed by the string and the terminator. A zero length string is represented by two bytes: 01 (string length) followed by the null terminator.

DosGetResource - Related Functions

Related Functions

- DosFreeResource
- DosLoadModule

DosGetResource - Example Code

This example loads the dynamic link module "DISPLAY.DLL," gets a resource object from it, and then releases it.

```
#define INCL_DOSRESOURCES     /* Resource types */
#define INCL_DOSMODULEMGR     /* Module Manager values */
#define INCL_DOSERRORS        /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR     LoadError[256] = "";         /* Area for Load failure information */
PSZ       ModuleName = "C:\\OS2\\DLL\\PMWP.DLL";  /* DLL with resources */
HMODULE   ModHandle  = NULLHANDLE;        /* Handle for module */
PVOID     ResPtr     = NULL;              /* Pointer to resource */
APIRET    rc         = NO_ERROR;          /* API return code */

 rc = DosLoadModule(LoadError,               /* Failure information buffer */
                    sizeof(LoadError),       /* Size of buffer             */
                    ModuleName,              /* Module to load             */
                    &ModHandle);             /* Module handle returned     */
 if (rc != NO_ERROR) {
    printf("DosLoadModule error: return code = %u\n", rc);
    return 1;
 }

 rc = DosGetResource(ModHandle,     /* Handle for DLL containing resources */
                     RT_POINTER,    /* Ask for  Pointer                    */
                     1L,            /*          with an ID of 1            */
                     &ResPtr);  /* Get back pointer                    */
 if (rc != NO_ERROR) {
    printf("DosGetResource error: return code = %u\n", rc);
    return 1;
 } else {
```

```
    printf("Resource pointer = 0x%x\n", ResPtr);
 } /* endif */

 rc = DosFreeResource(ResPtr);
 if (rc != NO_ERROR) {
    printf("DosFreeResource error: return code = %u\n", rc);
    return 1;
 }

return NO_ERROR;
}
```

DosGetResource - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGetSharedMem

DosGetSharedMem - Syntax

Obtains access to a shared memory object.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pb;    /*  The base virtual address of the gettable shared memory object as assigned by DosAllocSharedM
ULONG     flag;  /*  Access protection flags. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosGetSharedMem(pb, flag);
```

DosGetSharedMem Parameter - pb

pb (PVOID) - input
The base virtual address of the gettable shared memory object as assigned by DosAllocSharedMem.

DosGetSharedMem Parameter - flag

flag (ULONG) - input
Access protection flags.

A set of attribute flags that specify the desired access protection for the shared memory object.

Desired Access Protection

The following access protections are available:

1	PAG_READ Read access.
2	PAG_WRITE Write access.
4	PAG_EXECUTE Execute access to the committed pages in the shared memory object.
8	PAG_GUARD Access to the committed pages in the shared memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits, PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

DosGetSharedMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGetSharedMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
212	ERROR_LOCKED
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosGetSharedMem - Parameters

pb (PVOID) - input
The base virtual address of the gettable shared memory object as assigned by DosAllocSharedMem.

flag (ULONG) - input
Access protection flags.

A set of attribute flags that specify the desired access protection for the shared memory object.

Desired Access Protection

The following access protections are available:

1	PAG_READ Read access.

2 PAG_WRITE
Write access.

4 PAG_EXECUTE
Execute access to the committed pages in the shared memory object.

8 PAG_GUARD
Access to the committed pages in the shared memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits, PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

ulrc (APIRET) - returns
Return Code.

DosGetSharedMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
212	ERROR_LOCKED
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosGetSharedMem - Remarks

DosGetSharedMem obtains access to a shared memory object.

Getting access to a shared memory object allocates the virtual address (of the shared memory object) in the virtual-address space of the process.

The virtual address of the gettable shared memory object is the base address assigned when the gettable shared memory object was created. The creating and receiving processes must use some form of InterProcess Communication (IPC) to exchange this value.

The shared memory object specified by the virtual address must be gettable (that is, it must have been created using DosAllocSharedMem with the OBJ_GETTABLE attribute set).

Gettable shared memory objects are mapped at the same virtual address in all processes that obtain access to the shared memory object.

The desired access protection applied to committed pages must be compatible with the access protection granted to the shared memory object when it was created.

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

Error code 31 (ERROR_GEN_FAILURE) is returned when the segment's maximum reference count of 65535 is exceeded.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosGetSharedMem - Related Functions

Related Functions

- DosAllocSharedMem
- DosGetNamedSharedMem
- DosGiveSharedMem

DosGetSharedMem - Example Code

This example shows how to get access to a shared memory object.

```
#define INCL_DOSMEMMGR   /* DOS Memory Management */
#define INCL_DOSERRORS   /* DOS error values      */
#include <os2.h>
#include <stdio.h>

int main (USHORT argc, PCHAR argv[])
{
   PVOID  pvShrObject = NULL;      /* Pointer to shared memory object      */
   APIRET rc          = NO_ERROR;  /* Return code                          */

   /*********************************************************************/
   /* This example assumes that pvShrObject can be initialized here with   */
   /* a pointer to a shared memory object with the OBJ_GETTABLE attribute  */
   /* Normally this would be passed to this program via inter-process      */
   /* communication (IPC) - a pipe, a queue, a semaphore, the file system. */
   /*********************************************************************/

   rc = DosGetSharedMem(pvShrObject, PAG_WRITE | PAG_READ);

   if (rc != NO_ERROR) {
      printf("DosGetSharedMem error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosGetSharedMem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosGiveSharedMem

DosGiveSharedMem - Syntax

Gives another process access to a shared memory object.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pb;    /*  The base virtual address of the giveable shared memory object as assigned by DosAllocSharedM
PID       pid;   /*  The identifier of the target process that is to receive access to the shared memory object.
ULONG     flag;  /*  Access protection flags. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosGiveSharedMem(pb, pid, flag);
```

DosGiveSharedMem Parameter - pb

pb (PVOID) - input
The base virtual address of the giveable shared memory object as assigned by DosAllocSharedMem.

DosGiveSharedMem Parameter - pid

pid (PID) - input
The identifier of the target process that is to receive access to the shared memory object.

DosGiveSharedMem Parameter - flag

flag (ULONG) - input
Access protection flags.

A set of attribute flags that specify the desired access protection for the shared memory object.

Desired Access Protection

The following access protections are available:

1	PAG_READ Read access.
2	PAG_WRITE Write access.
4	PAG_EXECUTE Execute access to the committed pages in the shared memory object.
8	PAG_GUARD Access to the committed pages in the shared memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits, PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

DosGiveSharedMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosGiveSharedMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
212	ERROR_LOCKED
303	ERROR_INVALID_PROCID
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosGiveSharedMem - Parameters

pb (PVOID) - input
The base virtual address of the giveable shared memory object as assigned by DosAllocSharedMem.

pid (PID) - input
The identifier of the target process that is to receive access to the shared memory object.

flag (ULONG) - input
Access protection flags.

A set of attribute flags that specify the desired access protection for the shared memory object.

Desired Access Protection

The following access protections are available:

1	PAG_READ Read access.
2	PAG_WRITE Write access.
4	PAG_EXECUTE Execute access to the committed pages in the shared memory object.
8	PAG_GUARD Access to the committed pages in the shared memory object causes a "guard page entered" condition to be raised in the subject process.

At least one of the bits, PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

ulrc (APIRET) - returns
Return Code.

DosGiveSharedMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
212	ERROR_LOCKED

303	ERROR_INVALID_PROCID
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosGiveSharedMem - Remarks

DosGiveSharedMem gives another process access to a shared memory object.

Giving access to a shared memory object allocates the virtual address (of the shared memory object) in the virtual-address space of the target process. This is similar to the target process' performing a DosGetSharedMem operation on the specified shared memory object.

The virtual address of the giveable shared memory object is the base address assigned when the giveable shared memory object was created. The creating and receiving processes must use some form of InterProcess Communication (IPC) to exchange this value.

Giveable shared memory objects are mapped at the same virtual address in all processes that obtain access to the shared memory object.

The shared memory object specified by the virtual address must be giveable (that is, it must have been created with the OBJ_GIVEABLE attribute set on a call to DosAllocSharedMem).

The desired access protection applied to committed pages must be compatible with the access protection granted to the shared memory object when it was created.

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosGiveSharedMem - Related Functions

Related Functions

- DosAllocSharedMem
- DosGetNamedSharedMem
- DosGetSharedMem

DosGiveSharedMem - Example Code

This example shows how to give access to a shared memory object to another process.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
```

```
#include <string.h>

int main (USHORT argc, PCHAR argv[])
{
   PVOID  pvShrObject = NULL;      /* Pointer to shared memory object      */
   ULONG  ulObjSize   = 0;         /* Size of memory object                */
   PID    pidSharer   = 0;         /* Process ID for partner               */
   APIRET rc          = NO_ERROR;  /* Return code                          */

   /*********************************************************************/
   /* This example assumes that pidSharer can be initialized here with  */
   /* the process ID for the process to which we are giving the memory.  */
   /* Normally this would be passed to this program via inter-process    */
   /* communication (IPC) - a pipe, a queue, a semaphore, the file system. */
   /*********************************************************************/

   ulObjSize = 4100;          /* Will be rounded to a page boundary - 8192 */

   rc = DosAllocSharedMem(&pvShrObject,   /* Pointer to object pointer     */
                          NULL,           /* Unnamed memory                */
                          ulObjSize,      /* Desired size of object        */
                          OBJ_GIVEABLE | PAG_COMMIT | PAG_WRITE | PAG_READ );
   if (rc != NO_ERROR) {
      printf("DosAllocSharedMem error:  return code = %u\n",rc);
      return 1;
   }

   rc = DosGiveSharedMem(pvShrObject,             /* Object pointer     */
                         pidSharer,               /* Process to give to */
                         PAG_WRITE | PAG_READ);   /* Memory attributes  */
   if (rc != NO_ERROR) {
      printf("DosGetSharedMem error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosGiveSharedMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosInsertMessage

DosInsertMessage - Syntax

Inserts variable text-string information into a message.

```
#define INCL_DOSMISC
#include <os2.h>
```

```
PCHAR     *pTable;  /*  A pointer table. */
ULONG      cTable;  /*  The number of variable insertion text strings. */
PSZ        pszMsg;  /*  The address of the input message. */
ULONG      cbMsg;   /*  The length, in bytes, of the input message. */
PCHAR      pBuf;    /*  The address of the caller's buffer area where the system returns the requested message.
ULONG      cbBuf;   /*  The length, in bytes, of the caller's buffer area. */
PULONG     pcbMsg;  /*  The length, in bytes, of the updated message returned. */
APIRET     ulrc;    /*  Return Code. */

ulrc = DosInsertMessage(pTable, cTable, pszMsg,
         cbMsg, pBuf, cbBuf, pcbMsg);
```

DosInsertMessage Parameter - pTable

pTable (PCHAR *) - input
A pointer table.

Each pointer filets to an ASCIIZ string or a double-byte character-set (DBCS) string ending in nulls. A maximum of nine strings can be present.

DosInsertMessage Parameter - cTable

cTable (ULONG) - input
The number of variable insertion text strings.

Possible values range from 0 to 9. If *cTable* is 0, *pTable* is ignored.

DosInsertMessage Parameter - pszMsg

pszMsg (PSZ) - input
The address of the input message.

DosInsertMessage Parameter - cbMsg

cbMsg (ULONG) - input
The length, in bytes, of the input message.

DosInsertMessage Parameter - pBuf

pBuf (PCHAR) - output

The address of the caller's buffer area where the system returns the requested message.

If the message is too long to fit in the caller's buffer, then as much of the message text as possible is returned, with the appropriate error return code.

DosInsertMessage Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of the caller's buffer area.

DosInsertMessage Parameter - pcbMsg

pcbMsg (PULONG) - output
The length, in bytes, of the updated message returned.

DosInsertMessage Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosInsertMessage returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
316	ERROR_MR_MSG_TOO_LONG
320	ERROR_MR_INV_IVCOUNT

For a full list of error codes, see Errors.

DosInsertMessage - Parameters

pTable (PCHAR *) - input
A pointer table.

Each pointer filets to an ASCIIZ string or a double-byte character-set (DBCS) string ending in nulls. A maximum of nine strings can be present.

cTable (ULONG) - input
The number of variable insertion text strings.

Possible values range from 0 to 9. If *cTable* is 0, *pTable* is ignored.

pszMsg (PSZ) - input
The address of the input message.

cbMsg (ULONG) - input
The length, in bytes, of the input message.

pBuf (PCHAR) - output
The address of the caller's buffer area where the system returns the requested message.

If the message is too long to fit in the caller's buffer, then as much of the message text as possible is returned, with the appropriate error return code.

cbBuf (ULONG) - input
The length, in bytes, of the caller's buffer area.

pcbMsg (PULONG) - output
The length, in bytes, of the updated message returned.

ulrc (APIRET) - returns
Return Code.

DosInsertMessage returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
316	ERROR_MR_MSG_TOO_LONG
320	ERROR_MR_INV_IVCOUNT

For a full list of error codes, see Errors.

DosInsertMessage - Remarks

DosInsertMessage inserts variable text-string information into a message.

DosInsertMessage differs from DosGetMessage in that it does not retrieve a message. It is particularly useful when messages are loaded early, before actual insertion text strings are known.

If *cTable* is greater than 9, DosInsertMessage returns an error indicating that *cTable* is out of range. A default message also is placed into the caller's buffer. Refer to DosGetMessage for details about default messages.

If the numeric value of x in the %x sequence for %1-%9 is less than or equal to *cTable* , then text insertion, by substitution for %x, is performed for all occurrences of %x in the message. Otherwise, text insertion is ignored, and the %x sequence is returned in the message unchanged. Text insertion is performed for all text-strings defined by *cTable* and *pTable* .

Variable data insertion does not depend on blank character delimiters, nor are blanks automatically inserted.

DosInsertMessage - Related Functions

Related Functions

- DosGetMessage
- DosPutMessage
- DosQueryMessageCP

DosInsertMessage - Example Code

This example demonstrate the insertion of the variable text into a message.

```
#define INCL_DOSMISC    /* Miscellaneous values */
#define INCL_DOSERRORS  /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
int main(VOID) {
UCHAR   *IvTable[3] = {0};                  /* Table of variables to insert */
UCHAR   MsgInput[30] = "Processing for %1: %2 has %3. ";
UCHAR   DataArea[80]= "";                   /* Output message area */
ULONG   MsgLength   = 0;                    /* Length of returned message */
APIRET  rc          = 0;                    /* Return code */
int     LoopCtr     = 0;                    /* for loop counter */

IvTable[0] = "function";
IvTable[1] = "DosInsertMessage";
IvTable[2] = "started";

               /* Insert strings in proper variable fields */

rc = DosInsertMessage(IvTable,             /* Message insert pointer array */
                      3,                   /* Number of inserts */
                      MsgInput,            /* Input message */
                      strlen(MsgInput),    /* Length of input message */
                      DataArea,            /* Output area for message */
                      sizeof(DataArea),    /* Size of output area */
                      &MsgLength);        /* Length of output message created */
if (rc != NO_ERROR) {
   printf("DosInsertMessage error: return code = %u\n", rc);
   return 1;
}

printf("%s\n", DataArea);   /* Print the resulting message */

return NO_ERROR;
}
```

DosInsertMessage - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosKillProcess

DosKillProcess - Syntax

Flags a process to end, and returns the termination code to its parent (if any).

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     action;  /*  The processes to be flagged for ending. */
PID       pid;     /*  Process ID of the process or root process of the process tree to be flagged for ending. *
APIRET    ulrc;    /*  Return Code. */

ulrc = DosKillProcess(action, pid);
```

DosKillProcess Parameter - action

action (ULONG) - input
The processes to be flagged for ending.

The values of this field are shown in the following list:

0 DKP_PROCESSTREE
A process and all its descendant processes. The process must be either the current process, or it must have been directly created by the current process using DosExecPgm with a value of 2 (EXEC_ASYNCRESULT) for *execFlag* .

After the indicated process ends, its descendants are flagged for ending. The indicated process need not still be executing. If it has ended, its descendants are still flagged for ending.

1 DKP_PROCESS
Any process. Only the indicated process is flagged for ending.

DosKillProcess Parameter - pid

pid (PID) - input
Process ID of the process or root process of the process tree to be flagged for ending.

DosKillProcess Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosKillProcess returns one of the following values:

0	NO_ERROR
13	ERROR_INVALID_DATA
217	ERROR_ZOMBIE_PROCESS
303	ERROR_INVALID_PROCID
305	ERROR_NOT_DESCENDANT

For a full list of error codes, see Errors.

DosKillProcess - Parameters

action (ULONG) - input
The processes to be flagged for ending.

The values of this field are shown in the following list:

0 DKP_PROCESSTREE

A process and all its descendant processes. The process must be either the current process, or it must have been directly created by the current process using DosExecPgm with a value of 2 (EXEC_ASYNCRESULT) for *execFlag* .

After the indicated process ends, its descendants are flagged for ending. The indicated process need not still be executing. If it has ended, its descendants are still flagged for ending.

1 DKP_PROCESS
Any process. Only the indicated process is flagged for ending.

pid (PID) - input
Process ID of the process or root process of the process tree to be flagged for ending.

ulrc (APIRET) - returns
Return Code.

DosKillProcess returns one of the following values:

0	NO_ERROR
13	ERROR_INVALID_DATA
217	ERROR_ZOMBIE_PROCESS
303	ERROR_INVALID_PROCID
305	ERROR_NOT_DESCENDANT

For a full list of error codes, see Errors.

DosKillProcess - Remarks

DosKillProcess allows a process to send the KILLPROCESS exception to another process or group of processes. The default action of the system is to end each of the processes. A process may intercept this action by installing an exception handler for the KILLPROCESS exception (see DosSetExceptionHandler). In such a case, the program will ensure the integrity of its files, and then issue DosExit.

If there is no exception handler, or if no handler handles the exception, then DosKillProcess affects the process as if one of its threads has issued DosKillProcess for the entire process. All file buffers are written, and the handles opened by the process are closed. Any internal buffers managed by the program externally of the system are not written. An example of such a buffer is a C-language library internal character buffer.

The parent of the process gets the "Unintercepted DosKillProcess" termination code when it issues DosWaitChild.

The "ERROR_ZOMBIE_PROCESS" error code indicates that the specified process has ended, but its parent has not yet issued DosWaitChild to get its return code.

DosKillProcess - Related Functions

Related Functions

- DosExecPgm
- DosExit
- DosExitList
- DosKillThread
- DosWaitChild

DosKillProcess - Example Code

This example uses DosKillProc to kill itself. It creates and opens a queue as a way of getting the PID for the process. In the end, it sleeps for up to 45 seconds, to ensure that the process was killed by this API.

```
#define INCL_DOSQUEUES    /* DOS Queue values */
#define INCL_DOSPROCESS   /* DOS thread and process values */
```

```
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(USHORT argc, PCHAR argv[]) {

   PSZ         szQueueName  = "\\QUEUES\\OF\\DATA\\WAITING\\FOR\\SERVICE";
   HQUEUE      hqSpecialQue = NULLHANDLE; /* Queue handle                    */
   REQUESTDATA Request      = {0};        /* Reques */
   PID         pidOwner     = 0;
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateQueue(&hqSpecialQue,    /* Queue handle                    */
             QUE_FIFO |                  /* First-In First-Out order        */
             QUE_CONVERT_ADDRESS,        /* Convert 16-bit addresses to 32  */
             szQueueName);               /* Name of the queue to create     */

   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;       }

   rc = DosOpenQueue (&pidOwner,         /* PID of queue owner              */
                      &hqSpecialQue,     /* Handle for created queue        */
                      szQueueName);      /* Name of the queue to open       */

   if (rc!= NO_ERROR) {
      printf ("DosOpenQueue error: return code = %u\n", rc);
      return 1;       }

         /* Kill the queue owner (which is us) */

   rc = DosKillProcess(0, pidOwner);
   if (rc != NO_ERROR) {
      printf("DosKillProcess error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep(45000L);    /* Dead code */

   return NO_ERROR;
}
```

DosKillProcess - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosKillThread

DosKillThread - Syntax

Allows a thread to end another thread in the current process.

```
#define INCL_DOSPROCESS
#include <os2.h>

TID       tid;   /*  Thread identifier within the current process to be ended. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosKillThread(tid);
```

DosKillThread Parameter - tid

tid (TID) - input
Thread identifier within the current process to be ended.

DosKillThread Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosKillThread returns one of the following values:

0	NO_ERROR
170	ERROR_BUSY
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosKillThread - Parameters

tid (TID) - input
Thread identifier within the current process to be ended.

ulrc (APIRET) - returns
Return Code.

DosKillThread returns one of the following values:

0	NO_ERROR
170	ERROR_BUSY
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosKillThread - Remarks

DosKillThread allows a thread to end another thread in the current process.

DosKillThread returns to the caller without waiting for the ending thread to complete its termination processing.

You cannot use this function to end the current thread. If you use DosKillThread to end thread 1, the entire process ends. This is similar to issuing DosExit for thread 1.

If the thread to be ended is executing 16-bit code, or has been created by a 16-bit request, ERROR_BUSY is returned.

This function will not terminate a thread that is suspended. Instead the suspended thread will be terminated when it resumes execution. For this reason, you should not kill the main thread of an application if there are any secondary threads that are suspended.

Note: This function is very powerful and must be used with extreme caution. It should be used only when the state of the target thread is known.

DosKillThread - Related Functions

Related Functions

- DosCreateThread
- DosExitList
- DosExecPgm
- DosExit
- DosKillProcess
- DosResumeThread
- DosSuspendThread
- DosWaitChild
- DosWaitThread

DosKillThread - Example Code

This example shows how to kill a thread given its TID.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#define INCL_DOSERRORS      /* DOS error values          */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(USHORT argc, PCHAR argv[] ) {

   APIRET      rc          = NO_ERROR; /* Return code                        */
   TID         tidToKill   = 0;        /* Kill this thread                   */

   if ( argc < 2 ) {
      printf("kthread error:  Need to pass TID of thread to kill.\n");
      return 1;
   } else {
      tidToKill = (TID) atoi ( argv[1] );
   } /* endif */

   rc = DosKillThread ( tidToKill );   /* Kill specified thread */

   if (rc != NO_ERROR) {
      printf("DosKillThread error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosKillThread complete.\n");
   } /* endif */

   return NO_ERROR;
}
```

DosKillThread - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosLoadModule

DosLoadModule - Syntax

Loads a dynamic link module (DLL), and returns a handle for the module.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

PSZ         pszName;     /*  The address of a buffer into which the name of an object that contributed to the fa
ULONG       cbName;      /*  The length, in bytes, of the buffer described by pszName. */
PSZ         pszModname;  /*  The address of an ASCIIZ name string that contains the dynamic link module name. */
PHMODULE    phmod;       /*  Pointer to an HMODULE in which the handle for the dynamic link module is returned.
APIRET      ulrc;        /*  Return Code. */

ulrc = DosLoadModule(pszName, cbName, pszModname,
         phmod);
```

DosLoadModule Parameter - pszName

pszName (PSZ) - output
The address of a buffer into which the name of an object that contributed to the failure of DosLoadModule is to be placed.

The name of the object is usually the name of a dynamic link library that either could not be found or could not be loaded.

DosLoadModule Parameter - cbName

cbName (ULONG) - input
The length, in bytes, of the buffer described by *pszName* .

DosLoadModule Parameter - pszModname

pszModname (PSZ) - input
The address of an ASCIIZ name string that contains the dynamic link module name.

The file-name extension used for dynamic link libraries is .DLL.

When a request is made to load a module and a fully-qualified path is specified, the system loads that .DLL, if it exists. If a fully-qualified path is *not* specified, the system checks if the .DLL is already loaded. If it is loaded, that .DLL is the one that is used; otherwise, the system searches the paths in the LIBPATH string in the "CONFIG.SYS" file and uses the first instance of the specified .DLL it finds. If the current directory is not specified in the LIBPATH, the system does not check that directory to see if a different version exists. Consequently, if two processes started from different directories use the same DLL, but different versions of that DLL exist in both directories, the version of the DLL loaded by the first process is the one used by both processes.

DosLoadModule Parameter - phmod

phmod (PHMODULE) - output
Pointer to an HMODULE in which the handle for the dynamic link module is returned.

DosLoadModule Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosLoadModule returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
11	ERROR_BAD_FORMAT
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
95	ERROR_INTERRUPT
108	ERROR_DRIVE_LOCKED
123	ERROR_INVALID_NAME
127	ERROR_PROC_NOT_FOUND
180	ERROR_INVALID_SEGMENT_NUMBER
182	ERROR_INVALID_ORDINAL
190	ERROR_INVALID_MODULETYPE
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID (this error code is not valid in OS/2 Warp PowerPC Edition)
194	ERROR_ITERATED_DATA_EXCEEDS_64K
195	ERROR_INVALID_MINALLOCSIZE
196	ERROR_DYNLINK_FROM_INVALID_RING
198	ERROR_INVALID_SEGDPL
199	ERROR_AUTODATASEG_EXCEEDS_64K
201	ERROR_RELOCSRC_CHAIN_EXCEEDS_SEGLIMIT
206	ERROR_FILENAME_EXCED_RANGE
295	ERROR_INIT_ROUTINE_FAILED

For a full list of error codes, see Errors.

DosLoadModule - Parameters

pszName (PSZ) - output
The address of a buffer into which the name of an object that contributed to the failure of DosLoadModule is to be placed.

The name of the object is usually the name of a dynamic link library that either could not be found or could not be loaded.

cbName (ULONG) - input
The length, in bytes, of the buffer described by *pszName* .

pszModname (PSZ) - input
The address of an ASCIIZ name string that contains the dynamic link module name.

The file-name extension used for dynamic link libraries is .DLL.

When a request is made to load a module and a fully-qualified path is specified, the system loads that .DLL, if it exists. If a fully-qualified path is *not* specified, the system checks if the .DLL is already loaded. If it is loaded, that .DLL is the one that is used; otherwise, the system searches the paths in the LIBPATH string in the "CONFIG.SYS" file and uses the first instance of the specified .DLL it finds. If the current directory is not specified in the LIBPATH, the system does not check that directory to see if a different version exists. Consequently, if two processes started from different directories use the same DLL, but different versions of that DLL exist in both directories, the version of the DLL loaded by the first process is the one used by both processes.

phmod (PHMODULE) - output
Pointer to an HMODULE in which the handle for the dynamic link module is returned.

ulrc (APIRET) - returns
Return Code.

DosLoadModule returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
11	ERROR_BAD_FORMAT
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
95	ERROR_INTERRUPT
108	ERROR_DRIVE_LOCKED
123	ERROR_INVALID_NAME
127	ERROR_PROC_NOT_FOUND
180	ERROR_INVALID_SEGMENT_NUMBER
182	ERROR_INVALID_ORDINAL
190	ERROR_INVALID_MODULETYPE
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID (this error code is not valid in OS/2 Warp PowerPC Edition)
194	ERROR_ITERATED_DATA_EXCEEDS_64K
195	ERROR_INVALID_MINALLOCSIZE
196	ERROR_DYNLINK_FROM_INVALID_RING
198	ERROR_INVALID_SEGDPL
199	ERROR_AUTODATASEG_EXCEEDS_64K
201	ERROR_RELOCSRC_CHAIN_EXCEEDS_SEGLIMIT
206	ERROR_FILENAME_EXCED_RANGE
295	ERROR_INIT_ROUTINE_FAILED

For a full list of error codes, see Errors.

DosLoadModule - Remarks

DosLoadModule loads a dynamic link module, and returns a handle for the module.

If the file is an OS/2 dynamic link module, then the module is loaded, and a handle is returned. The returned handle is used for freeing the dynamic link module, getting procedure addresses, and getting the fully qualified file name.

DosLoadModule cannot be issued from ring 2 if the dynamic library routine has an initialization routine, or the process will be terminated.

If the module has an initialization routine that is in an object that has IOPL indicated, any process attempting to use the module will cause a general protection fault, and will be terminated.

DosLoadModule - Related Functions

Related Functions

- DosExecPgm
- DosFreeModule
- DosQueryModuleName
- DosQueryProcAddr

DosLoadModule - Example Code

This example loads the dynamic link module "DISPLAY.DDL," queries its address and type, and finally frees it.

```
 #define INCL_DOSMODULEMGR     /* Module Manager values */
 #define INCL_DOSERRORS        /* Error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

 PSZ      ModuleName     = "C:\\OS2\\DLL\\DISPLAY.DLL";  /* Name of module   */
 UCHAR    LoadError[256] = "";          /* Area for Load failure information */
 HMODULE  ModuleHandle   = NULLHANDLE;  /* Module handle                     */
 PFN      ModuleAddr     = 0;           /* Pointer to a system function      */
 ULONG    ModuleType     = 0;           /* Module type                       */
 APIRET   rc             = NO_ERROR;    /* Return code                       */

   rc = DosLoadModule(LoadError,               /* Failure information buffer */
                      sizeof(LoadError),       /* Size of buffer             */
                      ModuleName,              /* Module to load             */
                      &ModuleHandle);          /* Module handle returned     */
   if (rc != NO_ERROR) {
      printf("DosLoadModule error: return code = %u\n", rc);
      return 1;
   } else {
      printf("Module %s loaded.\n", ModuleName);
   } /* endif */

   rc = DosQueryProcAddr(ModuleHandle,         /* Handle to module           */
                         1L,                   /* No ProcName specified      */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleAddr);         /* Address returned           */
   if (rc != NO_ERROR) {
      printf("DosQueryProcAddr error: return code = %u\n", rc);
      return 1;
   } else printf("Address of module is 0x%x.\n", ModuleAddr);

   rc = DosQueryProcType(ModuleHandle,         /* Handle to module           */
                         1L,                   /* Indicate no ProcName given */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleType);         /* Type 0=16-bit   1=32-bit   */
   if (rc != NO_ERROR) {
      printf("DosQueryProcType error: return code = %u\n", rc);
      return 1;
   } else printf("This is a %s module.\n", ( ModuleType ? "32-bit" : "16-bit"));

   rc = DosFreeModule(ModuleHandle);
   if (rc != NO_ERROR) {
      printf("DosFreeModule error: return code = %u\n", rc);
```

```
      return 1;
   } else printf("Module %s freed.\n", ModuleName);

   return NO_ERROR;
}
```

DosLoadModule - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosMapCase

DosMapCase - Syntax

Performs case mapping on a string of binary values that represent ASCII characters.

```
#define INCL_DOSNLS
#include <os2.h>

ULONG           cb;    /*  The length, in bytes, of the string of binary values to be case-mapped. */
PCOUNTRYCODE    pcc;   /*  Pointer to the COUNTRYCODE structure in which the country code and code page are iden
PCHAR           pch;   /*  The string of binary characters to be case-mapped. */
APIRET          ulrc;  /*  Return Code. */

ulrc = DosMapCase(cb, pcc, pch);
```

DosMapCase Parameter - cb

cb (ULONG) - input
The length, in bytes, of the string of binary values to be case-mapped.

DosMapCase Parameter - pcc

pcc (PCOUNTRYCODE) - input
Pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the case map table for the default system country code is used.

If *codepage* is set to 0, the case map table for the current process code page of the caller is used.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

DosMapCase Parameter - pch

pch (PCHAR) - in/out
The string of binary characters to be case-mapped.

They are case-mapped in place, and they replace the input, so the results appear in *pch* .

DosMapCase Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosMapCase returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosMapCase - Parameters

cb (ULONG) - input
The length, in bytes, of the string of binary values to be case-mapped.

pcc (PCOUNTRYCODE) - input
Pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the case map table for the default system country code is used.

If *codepage* is set to 0, the case map table for the current process code page of the caller is used.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

pch (PCHAR) - in/out
The string of binary characters to be case-mapped.

They are case-mapped in place, and they replace the input, so the results appear in *pch* .

ulrc (APIRET) - returns
Return Code.

DosMapCase returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosMapCase - Remarks

DosMapCase performs case mapping on a string of binary values that represent ASCII characters.

The case map in the country file (the default name is COUNTRY.SYS) that corresponds to the system country code or selected country code, and to the process code page or selected code page, is used to perform the case mapping.

DosMapCase - Related Functions

Related Functions

- DosQueryCollate
- DosQueryCp
- DosQueryCtryInfo
- DosQueryDBCSEnv
- DosSetProcessCp

DosMapCase - Example Code

This example converts a string to uppercase based on the current country settings.

```
#define INCL_DOSNLS     /* DOS National Language Support values */
#define INCL_DOSERRORS  /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

#define COUNTRY_CODE 0       /* Country code             (0 = current) */
#define NLS_CODEPAGE 0       /* Code page for conversion (0 = current) */

int main(VOID) {

COUNTRYCODE Country       = {0};        /* Country code  */
CHAR        uchString[80] = "";         /* String        */
APIRET      rc            = 0;          /* Return code   */

   Country.country  = COUNTRY_CODE;     /* Country code  */

   Country.codepage = NLS_CODEPAGE;     /* Code page     */

   strcpy(uchString, "Capitalize this entire sTrInG, please!");

   printf("Original string is:  %s\n", uchString);

   rc = DosMapCase(sizeof(uchString),   /* Length of string to convert */
                   &Country,            /* Country and code page info  */
                   uchString);          /* String to convert           */

   if (rc != NO_ERROR) {
      printf("DosMapCase error: return code = %u\n", rc);
      return 1;
```

```
   } else {
      printf("Converted string is: %s\n", uchString);
   } /* endif */

   return NO_ERROR;
}
```

DosMapCase - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosMove

DosMove - Syntax

Moves a file object to another location, and changes its name.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszOld;  /*  Address of the old path name of the file or subdirectory to be moved. */
PSZ       pszNew;  /*  Address of the new path name of the file or subdirectory. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosMove(pszOld, pszNew);
```

DosMove Parameter - pszOld

pszOld (PSZ) - input
Address of the old path name of the file or subdirectory to be moved.

DosMove Parameter - pszNew

pszNew (PSZ) - input
Address of the new path name of the file or subdirectory.

DosMove Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosMove returns the one of following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
17	ERROR_NOT_SAME_DEVICE
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE
250	ERROR_CIRCULARITY_REQUESTED
251	ERROR_DIRECTORY_IN_CDS

For a full list of error codes, see Errors.

DosMove - Parameters

pszOld (PSZ) - input
Address of the old path name of the file or subdirectory to be moved.

pszNew (PSZ) - input
Address of the new path name of the file or subdirectory.

ulrc (APIRET) - returns
Return Code.

DosMove returns the one of following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
17	ERROR_NOT_SAME_DEVICE
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE
250	ERROR_CIRCULARITY_REQUESTED
251	ERROR_DIRECTORY_IN_CDS

For a full list of error codes, see Errors.

DosMove - Remarks

DosMove can be used to change only the name of a file or subdirectory, allowing the file object to remain in the same subdirectory. Global file-name characters are not allowed in the source or target name.

If the specified paths are different, the subdirectory location of the file object is changed also. If a drive is specified for the target, it must be the same as the one specified or implied by the source.

Attempts to move a parent subdirectory to one of its descendant subdirectories result in error code 251 (ERROR_DIRECTORY_IN_CDS) because a subdirectory cannot be both an ancestor and a descendant of the same subdirectory.

Attempts to move the current subdirectory or any of its ancestors for the current process, or any other process, will be rejected.

Attributes (times and dates) of the source file object are moved to the target. If read-only files exist in the target path, they are not replaced.

During initialization by an application, DosQuerySysInfo is called to determine the maximum path length allowed by the operating system.

DosMove can be used to change the case of a file on a drive that is controlled by a file system driver (FSD). The following example would change the name of the file to "File.Txt.".

```
DosMove("file.txt","File.Txt")
```

DosMove - Related Functions

Related Functions

- DosClose
- DosCopy
- DosDelete
- DosQuerySysInfo
- DosQueryCurrentDisk
- DosSetDefaultDisk

DosMove - Example Code

This example moves the file "FIRST.DAT" to the directory "NEWDIR", renamed as "SECOND.DAT".

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>

HFILE  hfFileHandle          = 0L;                /* File Handle             */
ULONG  ulAction              = 0;                 /* Action taken            */
UCHAR  uchNewDirName[10]     = "newdir";          /* New directory name      */
PEAOP2 peaop2NewDirAttribute = NULL;              /* New directory attributes */
UCHAR  uchOldPathName[40]    = "first.dat";       /* Old path name string    */
UCHAR  uchNewPathName[40]    = "newdir\\second.dat"; /* New path name string */
APIRET rc                    = NO_ERROR;          /* Return code             */

int main(VOID) {

   /* Create a file "first.dat" in the current directory */

   rc = DosOpen("first.dat", &hfFileHandle, &ulAction,
                100L, FILE_NORMAL, FILE_CREATE | OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, 0L);
   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;        }

   rc = DosClose(hfFileHandle);          /* Close the file (it contains junk) */
   if (rc != NO_ERROR) {
      printf("DosClose error: return code = %u\n", rc);
```

```
      return 1;        }

   /* Create a new subdirectory within the current directory */

   rc = DosCreateDir(uchNewDirName, peaop2NewDirAttribute);
   if (rc != NO_ERROR) {
     printf("DosCreateDir error: return code = %u\n", rc);
     return 1;         }

   /* Move the file "first.dat" from the current directory to
      the new directory "newdir", and rename it "second.dat" */

   rc = DosMove(uchOldPathName, uchNewPathName);
   if (rc != NO_ERROR) {
     printf("DosMove error: return code = %u\n", rc);
     return 1;
   } else {
     printf("DosMove:  Move from %s to %s complete.\n",
             uchOldPathName, uchNewPathName); }

 return NO_ERROR;
}
```

DosMove - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosOpen

DosOpen - Syntax

Opens a new file, an existing file, or a replacement for an existing file. An open file can have extended attributes.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszFileName;  /*  Address of the ASCIIZ path name of the file or device to be opened. */
PHFILE    pHf;          /*  Address of the handle for the file. */
PULONG    pulAction;    /*  Address of the variable that receives the value that specifies the action taken by t
ULONG     cbFile;       /*  New logical size of the file (end of data, EOD), in bytes. */
ULONG     ulAttribute;  /*  File attribute information. */
ULONG     fsOpenFlags;  /*  The action to be taken depending on whether the file exists or does not exist. */
ULONG     fsOpenMode;   /*  The mode of the open function. Possible values are shown in the following list: */
PEAOP2    peaop2;       /*  Extended attributes. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosOpen(pszFileName, pHf, pulAction,
         cbFile, ulAttribute, fsOpenFlags,
         fsOpenMode, peaop2);
```

DosOpen Parameter - pszFileName

pszFileName (PSZ) - input
Address of the ASCIIZ path name of the file or device to be opened.

DosOpen Parameter - pHf

pHf (PHFILE) - output
Address of the handle for the file.

DosOpen Parameter - pulAction

pulAction (PULONG) - output
Address of the variable that receives the value that specifies the action taken by the DosOpen function.

If DosOpen fails, this value has no meaning. Otherwise, it is one of the following values:

1	FILE_EXISTED File already existed.
2	FILE_CREATED File was created.
3	FILE_TRUNCATED File existed and was changed to a given size (file was replaced).

DosOpen Parameter - cbFile

cbFile (ULONG) - input
New logical size of the file (end of data, EOD), in bytes.

This parameter is significant only when creating a new file or replacing an existing one. Otherwise, it is ignored. It is an error to create or replace a file with a nonzero length if the *fsOpenMode* Access-Mode flag is set to read-only.

DosOpen Parameter - ulAttribute

ulAttribute (ULONG) - input
File attribute information.

Possible values are shown in the following list.:

Bit	Description
31-6	Reserved, must be 0.
5	FILE_ARCHIVED (0x00000020) File has been archived.
4	FILE_DIRECTORY (0x00000010) File is a subdirectory.
3	Reserved, must be 0.
2	FILE_SYSTEM (0x00000004) File is a system file.
1	FILE_HIDDEN (0x00000002) File is hidden and does not appear in a directory listing.
0	FILE_READONLY (0x00000001) File can be read from, but not written to.
0	FILE_NORMAL (0x00000000) File can be read from or written to.

File attributes apply only if the file is created.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates a read-only file that has been archived.

DosOpen Parameter - fsOpenFlags

fsOpenFlags (ULONG) - input

The action to be taken depending on whether the file exists or does not exist.

Possible values are shown in the following list:

Bits	Description	
31-8	Reserved, must be 0.	
7-4	The following flags apply if the file does not exist:	
	0000	OPEN_ACTION_FAIL_IF_NEW Open an existing file; fail if the file does not exist.
	0001	OPEN_ACTION_CREATE_IF_NEW Create the file if the file does not exist.
3-0	The following flags apply if the file already exists:	
	0000	OPEN_ACTION_FAIL_IF_EXISTS Open the file; fail if the file already exists.
	0001	OPEN_ACTION_OPEN_IF_EXISTS Open the file if it already exists.
	0010	OPEN_ACTION_REPLACE_IF_EXISTS Replace the file if it already exists.

DosOpen Parameter - fsOpenMode

fsOpenMode (ULONG) - input
The mode of the open function. Possible values are shown in the following list:

Bit	Description
31-16	Reserved, must be zero.
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag: 0 — *pszFileName* represents a file to be opened normally. 1 — *pszFileName* is "drive:" (such as C: or A:), and represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Writes to the file may go through the file-system driver's cache. The file-system driver writes the sectors when the cache is full or the file is closed. 1 — Writes to the file may go through the file-system driver's cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous write operations. This bit flag is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always get reported directly to the caller by way of return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O calls. This flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00001000) No-Cache/Cache flag: 0 — The file-system driver should place data from I/O operations into its cache. 1 — I/O operations to the file need not be done through the file-system driver's cache. The setting of this bit determines whether file-system drivers should place data into the cache. Like the write-through bit, this is a per-handle bit, and is not inherited by child processes.
11	Reserved; must be 0.
10-8	The locality of reference flags contain information about how the application is to get access to the file. The values are as follows: 000 — OPEN_FLAGS_NO_LOCALITY (0x00000000) No locality known. 001 — OPEN_FLAGS_SEQUENTIAL (0x00000100) Mainly sequential access. 010 — OPEN_FLAGS_RANDOM (0x00000200) Mainly random access. 011 — OPEN_FLAGS_RANDOMSEQUENTIAL (0x00000300) Random with some locality.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 — File handle is inherited by a process created from a call to DosExecPgm. 1 — File handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing Mode flags. This field defines any restrictions to file access placed by the caller on other processes. The values are as follows:

001	OPEN_SHARE_DENYREADWRITE (0x00000010) Deny read/write access.
010	OPEN_SHARE_DENYWRITE (0x00000020) Deny write access.
011	OPEN_SHARE_DENYREAD (0x00000030) Deny read access.
100	OPEN_SHARE_DENYNONE (0x00000040) Deny neither read nor write access (deny none).

Any other value is invalid.

3 Reserved; must be 0.

2-0 Access-Mode flags. This field defines the file access required by the caller. The values are as follows:

000	OPEN_ACCESS_READONLY (0x00000000) Read-only access
001	OPEN_ACCESS_WRITEONLY (0x00000001) Write-only access
010	OPEN_ACCESS_READWRITE (0x00000002) Read/write access.

Any other value is invalid, as are any other combinations.

File sharing requires the cooperation of sharing processes. This cooperation is communicated through sharing and access modes. Any sharing restrictions placed on a file opened by a process are removed when the process closes the file with a DosClose request.

Sharing Mode

Specifies the type of file access that other processes may have. For example, if other processes can continue to read the file while your process is operating on it, specify Deny Write. The sharing mode prevents other processes from writing to the file but still allows them to read it.

Access Mode

Specifies the type of file access (access mode) needed by your process. For example, if your process requires read/write access, and another process has already opened the file with a sharing mode of Deny None, your DosOpen request succeeds. However, if the file is open with a sharing mode of Deny Write, the process is denied access.

If the file is inherited by a child process, all sharing and access restrictions also are inherited.

If an open file handle is duplicated by a call to DosDupHandle, all sharing and access restrictions also are duplicated.

DosOpen Parameter - peaop2

peaop2 (PEAOP2) - in/out
Extended attributes.

This parameter is only used to specify extended attributes (EAs) when creating a new file, replacing an existing file, or truncating an existing file. When opening existing files, it should be set to null.

Input	The address of the extended-attribute buffer, which contains an EAOP2 structure. *fpFEA2List* points to a data area where the relevant FEA2 list is to be found. *fpGEA2List* and *oError* are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code from DosOpen is the error code for that error condition. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the file.

If extended attributes are not to be defined or modified, the pointer *peaop2* must be set to zero.

DosOpen Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosOpen returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

For a full list of error codes, see Errors.

DosOpen - Parameters

pszFileName (PSZ) - input
Address of the ASCIIZ path name of the file or device to be opened.

pHf (PHFILE) - output
Address of the handle for the file. A file handle of zero is valid.

pulAction (PULONG) - output
Address of the variable that receives the value that specifies the action taken by the DosOpen function.

If DosOpen fails, this value has no meaning. Otherwise, it is one of the following values:

1	FILE_EXISTED File already existed.
2	FILE_CREATED File was created.
3	FILE_TRUNCATED File existed and was changed to a given size (file was replaced).

cbFile (ULONG) - input
New logical size of the file (end of data, EOD), in bytes.

This parameter is significant only when creating a new file or replacing an existing one. Otherwise, it is ignored. It is an error to create or replace a file with a nonzero length if the *fsOpenMode* Access-Mode flag is set to read-only.

ulAttribute (ULONG) - input
File attribute information.

Possible values are shown in the following list.:

Bit	Description
31-6	Reserved, must be 0.
5	FILE_ARCHIVED (0x00000020) File has been archived.
4	FILE_DIRECTORY (0x00000010) File is a subdirectory.
3	Reserved, must be 0.
2	FILE_SYSTEM (0x00000004) File is a system file.
1	FILE_HIDDEN (0x00000002) File is hidden and does not appear in a directory listing.
0	FILE_READONLY (0x00000001) File can be read from, but not written to.
0	FILE_NORMAL (0x00000000) File can be read from or written to.

File attributes apply only if the file is created.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates a read-only file that has been archived.

fsOpenFlags (ULONG) - input
The action to be taken depending on whether the file exists or does not exist.

Possible values are shown in the following list:

Bits	Description	
31-8	Reserved, must be 0.	
7-4	The following flags apply if the file does not exist:	
	0000	OPEN_ACTION_FAIL_IF_NEW Open an existing file; fail if the file does not exist.
	0001	OPEN_ACTION_CREATE_IF_NEW Create the file if the file does not exist.
3-0	The following flags apply if the file already exists:	
	0000	OPEN_ACTION_FAIL_IF_EXISTS Open the file; fail if the file already exists.
	0001	OPEN_ACTION_OPEN_IF_EXISTS Open the file if it already exists.
	0010	OPEN_ACTION_REPLACE_IF_EXISTS Replace the file if it already exists.

fsOpenMode (ULONG) - input
The mode of the open function. Possible values are shown in the following list:

Bit	Description	
31-16	Reserved, must be zero.	
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag:	
	0	*pszFileName* represents a file to be opened normally.
	1	*pszFileName* is "drive:" (such as C: or A:), and represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000)	

Write-Through flag:

0 Writes to the file may go through the file-system driver's cache. The file-system driver writes the sectors when the cache is full or the file is closed.

1 Writes to the file may go through the file-system driver's cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous write operations. This bit flag is not inherited by child processes.

13 OPEN_FLAGS_FAIL_ON_ERROR (0x00002000)
Fail-Errors flag. Media I/O errors are handled as follows:

0 Reported through the system critical-error handler.
1 Reported directly to the caller by way of a return code.

Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always get reported directly to the caller by way of return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O calls.

This flag bit is not inherited by child processes.

12 OPEN_FLAGS_NO_CACHE (0x00001000)
No-Cache/Cache flag:

0 The file-system driver should place data from I/O operations into its cache.

1 I/O operations to the file need not be done through the file-system driver's cache.

The setting of this bit determines whether file-system drivers should place data into the cache. Like the write-through bit, this is a per-handle bit, and is not inherited by child processes.

11 Reserved; must be 0.

10-8 The locality of reference flags contain information about how the application is to get access to the file. The values are as follows:

000 OPEN_FLAGS_NO_LOCALITY (0x00000000)
No locality known.

001 OPEN_FLAGS_SEQUENTIAL (0x00000100)
Mainly sequential access.

010 OPEN_FLAGS_RANDOM (0x00000200)
Mainly random access.

011 OPEN_FLAGS_RANDOMSEQUENTIAL (0x00000300)
Random with some locality.

7 OPEN_FLAGS_NOINHERIT (0x00000080)
Inheritance flag:

0 File handle is inherited by a process created from a call to DosExecPgm.

1 File handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing Mode flags. This field defines any restrictions to file access placed by the caller on other processes. The values are as follows:

001 OPEN_SHARE_DENYREADWRITE (0x00000010)
Deny read/write access.

010 OPEN_SHARE_DENYWRITE (0x00000020)
Deny write access.

011 OPEN_SHARE_DENYREAD (0x00000030)
Deny read access.

100 OPEN_SHARE_DENYNONE (0x00000040)
Deny neither read nor write access (deny none).

Any other value is invalid.

3 Reserved; must be 0.

2-0 Access-Mode flags. This field defines the file access required by the caller. The values are as follows:

000	OPEN_ACCESS_READONLY (0x00000000) Read-only access
001	OPEN_ACCESS_WRITEONLY (0x00000001) Write-only access
010	OPEN_ACCESS_READWRITE (0x00000002) Read/write access.

Any other value is invalid, as are any other combinations.

File sharing requires the cooperation of sharing processes. This cooperation is communicated through sharing and access modes. Any sharing restrictions placed on a file opened by a process are removed when the process closes the file with a DosClose request.

Sharing Mode

Specifies the type of file access that other processes may have. For example, if other processes can continue to read the file while your process is operating on it, specify Deny Write. The sharing mode prevents other processes from writing to the file but still allows them to read it.

Access Mode

Specifies the type of file access (access mode) needed by your process. For example, if your process requires read/write access, and another process has already opened the file with a sharing mode of Deny None, your DosOpen request succeeds. However, if the file is open with a sharing mode of Deny Write, the process is denied access.

If the file is inherited by a child process, all sharing and access restrictions also are inherited.

If an open file handle is duplicated by a call to DosDupHandle, all sharing and access restrictions also are duplicated.

peaop2 (PEAOP2) - in/out
Extended attributes.

This parameter is only used to specify extended attributes (EAs) when creating a new file, replacing an existing file, or truncating an existing file. When opening existing files, it should be set to null.

Input The address of the extended-attribute buffer, which contains an EAOP2 structure. *fpFEA2List* points to a data area where the relevant FEA2 list is to be found. *fpGEA2List* and *oError* are ignored.

Output *fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code from DosOpen is the error code for that error condition. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the file.

If extended attributes are not to be defined or modified, the pointer *peaop2* must be set to zero.

ulrc (APIRET) - returns
Return Code.

DosOpen returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

For a full list of error codes, see Errors.

DosOpen - Remarks

A successful DosOpen request returns a handle for accessing the file. A file handle of zero is valid. The read/write pointer is set at the first byte of the file. The position of the pointer can be changed with DosSetFilePtr or by read and write operations on the file.

The file's date and time can be queried with DosQueryFileInfo. They are set with DosSetFileInfo.

The read-only attribute of a file can be set with the ATTRIB command.

ulAttribute cannot be set to Volume Label. To set volume-label information, issue DosSetFSInfo with a logical drive number. Volume labels cannot be opened.

cbFile affects the size of the file only when the file is new or a is replacement. If an existing file is opened, *cbFile* is ignored. To change the size of the existing file, issue DosSetFileSize.

The value in *cbFile* is a recommended size. If the full size cannot be allocated, the open request may still succeed. The file system makes a reasonable attempt to allocate the new size in an area that is as nearly contiguous as possible on the medium. When the file size is extended, the values of the new bytes are undefined.

The Direct Open bit provides direct access to an entire disk or diskette volume, independent of the file system. This mode of opening the volume that is currently on the drive returns a handle to the calling function; the handle represents the logical volume as a single file. The calling function specifies this handle with a DosDevIOCtl Category 8, DSK_LOCKDRIVE request to prevent other processes from accessing the logical volume. When you are finished using the logical volume, issue a DosDevIOCtl Category 8, DSK_UNLOCKDRIVE request to allow other processes to access the logical volume.

The file-handle state bits can be set by DosOpen and DosSetFHState. An application can query the file-handle state bits, as well as the rest of the Open Mode field, by issuing DosQueryFHState.

You can use an EAOP2 structure to set extended attributes in *peaop2* when creating a file, replacing an existing file, or truncating an existing file. No extended attributes are set when an existing file is just opened.

A replacement operation is logically equivalent to atomically deleting and re-creating the file. This means that any extended attributes associated with the file also are deleted before the file is re-created.

DosOpen - Related Functions

Related Functions

- DosClose
- DosDevIOCtl
- DosDupHandle
- DosQueryHType
- DosSetFileInfo
- DosSetFilePtr
- DosSetFileSize
- DosSetMaxFH
- DosSetRelMaxFH

DosOpen - Example Code

This example opens or creates and opens a normal file named "DOSTEST.DAT", writes to it, reads from it, and finally closes it.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
```

```
#include <string.h>

int main(void) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  ulBytesRead    = 0;      /* Number of bytes read by DosRead */
   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   ULONG  ulLocal        = 0;      /* File pointer position after DosSetFilePtr */
   UCHAR  uchFileName[20]  = "dostest.dat",     /* Name of file */
          uchFileData[100] = " ";               /* Data to write to file */
   APIRET rc             = NO_ERROR;           /* Return code */

   /* Open the file test.dat.  Use an existing file or create a new */
   /* one if it doesn't exist.                                      */
   rc = DosOpen(uchFileName,                    /* File path name */
                &hfFileHandle,                  /* File handle */
                &ulAction,                      /* Action taken */
                100L,                           /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
                OPEN_ACTION_CREATE_IF_NEW |
                OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type */
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file */
                0L);                            /* No extended attribute */

   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosOpen: Action taken = %ld\n", ulAction);
   } /* endif */
   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n1...\n2...\n3\n");

   rc = DosWrite (hfFileHandle,                 /* File handle */
                  (PVOID) uchFileData,          /* String to be written */
                  sizeof (uchFileData),         /* Size of string to be written */
                  &ulWrote);                    /* Bytes actually written */

   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosSetFilePtr (hfFileHandle,            /* File Handle */
                       0L,                      /* Offset */
                       FILE_BEGIN,              /* Move from BOF */
                       &ulLocal);               /* New location address */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosRead (hfFileHandle,                 /* File Handle */
                 uchFileData,                  /* String to be read */
                 100L,                         /* Length of string to be read */
                 &ulBytesRead);                /* Bytes actually read */

   if (rc != NO_ERROR) {
      printf("DosRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosClose(hfFileHandle);                 /* Close the file */

   if (rc != NO_ERROR) {
      printf("DosClose error: return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
}
```

The following example shows how to open a communications port:

```
#define INCL_DOSFILEMGR          /* File Manager values      */
#define INCL_DOSERRORS           /* DOS Error values         */
#define INCL_DOSPROCESS          /* DOS Process values       */
#define INCL_DOSMISC             /* DOS Miscellanous values  */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(void) {

PSZ      pszCommPort    = "COM1";     /* Port name, could use "\\DEV\COM1"  */
HFILE    hPort          = NULLHANDLE; /* Handle for accessing port          */
ULONG    ulAction       = 0L;         /* DosOpen action                     */
ULONG    ulWrote        = 0;          /* Number of bytes written to port    */
UCHAR    uchPortData[100] = " ";      /* Data to write to port              */
APIRET   rc             = NO_ERROR;   /* Return code                        */
DosError( FERR_DISABLEHARDERR);       /* Disable hard error pop-up messages */
rc = DosOpen( pszCommPort,            /* Communications port to open        */
              &hPort,
              &ulAction,              /* Returns action taken by DosOpen    */
              0L,                     /* Not needed for byte stream devices */
              FILE_NORMAL,
              OPEN_ACTION_OPEN_IF_EXISTS,
              OPEN_ACCESS_READWRITE |
              OPEN_FLAGS_NOINHERIT  |
              OPEN_SHARE_DENYREADWRITE ,  /* Prevents us from opening port      */
                                          /* if another application is using it */
                                          /* and prevents other applications    */
                                          /* from using port while we have it   */
              0L);                        /* No extended attributes             */
DosError( FERR_ENABLEHARDERR);            /* Re-enable hard error pop-ups       */


if (rc != NO_ERROR) {
  printf("DosOpen error: return code = %u\n", rc);
  return 1;
} else {
  printf ("DosOpen: Action taken = %ld\n", ulAction);
} /* endif */
```

DosOpen - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosOpenEventSem

DosOpenEventSem - Syntax

Opens an event semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

PSZ       pszName;  /*  A pointer to the ASCIIZ name of the semaphore to open. */
PHEV      phev;     /*  A pointer to the event-semaphore handle. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosOpenEventSem(pszName, phev);
```

DosOpenEventSem Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore to open.

This field is null if the semaphore is either an unnamed, shared event semaphore or a private event semaphore (private semaphores are always unnamed). An unnamed event semaphore is identified by the pointer to the event semaphore handle (*phev*). If this field is not null, then the semaphore is a named shared semaphore, and *phev* must be set to zero.

DosOpenEventSem Parameter - phev

phev (PHEV) - in/out
A pointer to the event-semaphore handle.

Input	A pointer to the event-semaphore handle to open if *pszName* is null. If *pszName* is not null, set *phev* to zero.
Output	A pointer to the event-semaphore handle that was opened.

DosOpenEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosOpenEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	INVALID_NAME
187	ERROR_SEM_NOT_FOUND
291	ERROR_TOO_MANY_OPENS

For a full list of error codes, see Errors.

DosOpenEventSem - Parameters

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore to open.

This field is null if the semaphore is either an unnamed, shared event semaphore or a private event semaphore (private semaphores are always unnamed). An unnamed event semaphore is identified by the pointer to the event semaphore handle (*phev*). If this field is not null, then the semaphore is a named shared semaphore, and *phev* must be set to zero.

phev (PHEV) - in/out
A pointer to the event-semaphore handle.

Input	A pointer to the event-semaphore handle to open if *pszName* is null. If *pszName* is not null, set *phev* to zero.
Output	A pointer to the event-semaphore handle that was opened.

ulrc (APIRET) - returns
Return Code.

DosOpenEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	INVALID_NAME
187	ERROR_SEM_NOT_FOUND
291	ERROR_TOO_MANY_OPENS

For a full list of error codes, see Errors.

DosOpenEventSem - Remarks

DosOpenEventSem opens (obtains access to) an event semaphore for all of the threads in the calling process.

Note: The process that created the semaphore has immediate access to the semaphore, and does not need to call DosOpenEventSem.

DosOpenEventSem - Related Functions

Related Functions

- DosCloseEventSem
- DosCreateEventSem
- DosPostEventSem
- DosQueryEventSem
- DosResetEventSem
- DosWaitEventSem

DosOpenEventSem - Example Code

This example handles the client side of a pipe. It opens an existing named pipe and event semaphore, sends a message to the host, displays the state of the pipe semaphore, reads the host reply, and finally closes the event semaphore and pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   HEV      hev                   = NULLHANDLE; /* Event semaphore handle */
   PIPEINFO PipeBuffer[4]         = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   int      i                     = 0;
   PIPESEMSTATE infobuf[3]        = {{0}};

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to Pipe.\n");

   rc = DosOpenEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev);
   if (rc != NO_ERROR) {
      printf("DosOpenEventSem error: error code = %u\n", rc);
      return 1;
   }

   printf("Enter message to send to PIPEHOST: ");
   fflush(NULL);  /* Flush printf buffer */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }

   rc = DosQueryNPipeSemState((HSEM) hev, infobuf, sizeof(PIPESEMSTATE)*3);
   if (rc != NO_ERROR) {
     printf("DosQueryNPipeSemState error: return code = %u\n",rc);
     return 1;
   }

   printf("Status  Flag   Key   Avail\n------ ------ ------ ------\n");

   for (i=0; i<3; i++)
     printf("%6u %6u %6u %6u\n", infobuf[i].fStatus,
            infobuf[i].fFlag, infobuf[i].usKey, infobuf[i].usAvail);

   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST: %s\n\n", message);

   rc = DosCloseEventSem(hev);
   /* Should check if (rc != NO_ERROR) here... */

   rc = DosClose(PipeHandle);
   /* Should check if (rc != NO_ERROR) here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosOpenEventSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosOpenMutexSem

DosOpenMutexSem - Syntax

Opens a mutex semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

PSZ       pszName;  /*  A pointer to the ASCIIZ name of the semaphore to open. */
PHMTX     phmtx;    /*  Pointer to a mutex-semaphore handle. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosOpenMutexSem(pszName, phmtx);
```

DosOpenMutexSem Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore to open.

This field is null if the semaphore is either an unnamed, shared mutex semaphore or a private mutex semaphore (private semaphores are always unnamed). An unnamed mutex semaphore is identified by the pointer to the mutex-semaphore handle (*phmtx*) If this field is not null, the semaphore is a named shared semaphore, and *phmtx* must be set to zero before the call is made.

DosOpenMutexSem Parameter - phmtx

phmtx (PHMTX) - in/out
Pointer to a mutex-semaphore handle.

Input	A pointer to the mutex-semaphore handle to open if *pszName* is null; otherwise, this field is set to zero.
Output	A pointer to the mutex-semaphore handle that was opened.

DosOpenMutexSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosOpenMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED
123	ERROR_INVALID_NAME
187	ERROR_SEM_NOT_FOUND
291	ERROR_TOO_MANY_OPENS

For a full list of error codes, see Errors.

DosOpenMutexSem - Parameters

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore to open.

This field is null if the semaphore is either an unnamed, shared mutex semaphore or a private mutex semaphore (private semaphores are always unnamed). An unnamed mutex semaphore is identified by the pointer to the mutex-semaphore handle (*phmtx*) If this field is not null, the semaphore is a named shared semaphore, and *phmtx* must be set to zero before the call is made.

phmtx (PHMTX) - in/out
Pointer to a mutex-semaphore handle.

Input	A pointer to the mutex-semaphore handle to open if *pszName* is null; otherwise, this field is set to zero.
Output	A pointer to the mutex-semaphore handle that was opened.

ulrc (APIRET) - returns
Return Code.

DosOpenMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED
123	ERROR_INVALID_NAME
187	ERROR_SEM_NOT_FOUND
291	ERROR_TOO_MANY_OPENS

For a full list of error codes, see Errors.

DosOpenMutexSem - Remarks

DosOpenMutexSem opens (obtains access to). a mutual exclusion (mutex) semaphore for all of the threads in the calling process.

Note: The process that created the semaphore has immediate access to the semaphore, and does not need to call DosOpenMutexSem.

DosOpenMutexSem - Related Functions

Related Functions

- DosCloseMutexSem
- DosCreateMutexSem
- DosQueryMutexSem
- DosReleaseMutexSem
- DosRequestMutexSem

DosOpenMutexSem - Example Code

This example shows how to create, open, request, query, release, and close a mutual exclusion (mutex) semaphore.

```
#define INCL_DOSSEMAPHORES      /* Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HMTX    hmtx     = NULLHANDLE; /* Mutex semaphore handle */
PID     pidOwner = 0;          /* PID of current mutex semaphore owner */
TID     tidOwner = 0;          /* TID of current mutex semaphore owner */
ULONG   ulCount  = 0;          /* Request count for the semaphore */
APIRET  rc       = NO_ERROR;   /* Return code */

   rc = DosCreateMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                          &hmtx, 0, FALSE);       /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }
        /* This would normally be done by another unit of work */
   rc = DosOpenMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                        &hmtx);                 /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosRequestMutexSem(hmtx,                      /* Handle of semaphore */
                           (ULONG) SEM_INDEFINITE_WAIT);  /* Timeout (none) */
   if (rc != NO_ERROR) {
      printf("DosRequestMutexSem error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryMutexSem(hmtx,         /* Handle of semaphore */
                         &pidOwner,    /* Process ID of owner */
                         &tidOwner,    /* Thread ID of owner */
                         &ulCount);    /* Count */
   if (rc != NO_ERROR) {
      printf("DosQueryMutexSem error: return code = %u\n", rc);
      return 1;
   } else  {
      printf("Semaphore owned by PID %u, TID %u.", pidOwner, tidOwner);
      printf("  Request count is %u.\n", ulCount);
   } /* endif */

   rc = DosReleaseMutexSem(hmtx);        /* Relinquish ownership */
   if (rc != NO_ERROR) {
      printf("DosReleaseMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosCloseMutexSem(hmtx);          /* Close mutex semaphore */
   if (rc != NO_ERROR) {
      printf("DosCloseMutexSem error: return code = %u\n", rc);
      return 1;
```

```
   }

return NO_ERROR;
}
```

DosOpenMutexSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosOpenMuxWaitSem

DosOpenMuxWaitSem - Syntax

Opens a muxwait semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

PSZ       pszName;  /*  A pointer to the ASCIIZ name of the semaphore to open. */
PHMUX     phmux;    /*  Pointer to a muxwait-semaphore handle. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosOpenMuxWaitSem(pszName, phmux);
```

DosOpenMuxWaitSem Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore to open.

If the semaphore is either a private or shared unnamed muxwait semaphore, this field must be null and the semaphore is identified by the pointer to the muxwait-semaphore handle (*phmux*).

If the semaphore is a shared named semaphore, this field is no null and *phmux* must be set to 0. (Private semaphores are always unnamed).

DosOpenMuxWaitSem Parameter - phmux

phmux (PHMUX) - in/out
Pointer to a muxwait-semaphore handle.

Input	A pointer to the muxwait-semaphore handle to open if *pszName* is null; otherwise, when using *pszName* , this field is set to zero.
Output	A pointer to the muxwait-semaphore handle that was opened.

DosOpenMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosOpenMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED
123	ERROR_INVALID_NAME
187	ERROR_SEM_NOT_FOUND
291	ERROR_TOO_MANY_OPENS

For a full list of error codes, see Errors.

DosOpenMuxWaitSem - Parameters

pszName (PSZ) - input
A pointer to the ASCIIZ name of the semaphore to open.

If the semaphore is either a private or shared unnamed muxwait semaphore, this field must be null and the semaphore is identified by the pointer to the muxwait-semaphore handle (*phmux*).

If the semaphore is a shared named semaphore, this field is no null and *phmux* must be set to 0. (Private semaphores are always unnamed).

phmux (PHMUX) - in/out
Pointer to a muxwait-semaphore handle.

Input	A pointer to the muxwait-semaphore handle to open if *pszName* is null; otherwise, when using *pszName* , this field is set to zero.
Output	A pointer to the muxwait-semaphore handle that was opened.

ulrc (APIRET) - returns
Return Code.

DosOpenMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED
123	ERROR_INVALID_NAME
187	ERROR_SEM_NOT_FOUND

For a full list of error codes, see Errors.

DosOpenMuxWaitSem - Remarks

When the parameter *pszName* is null, *phmux* must point to have a valid handle. This handle can be created through DosCreateMuxWaitSem or through the situation described in this function.

DosOpenMuxWaitSem opens (obtains access to) a multiple wait (muxwait) semaphore for all of the threads in the calling process.

Note: The process that created the semaphore has immediate access to the semaphore, and does not need to call DosOpenMuxWaitSem.

DosOpenMuxWaitSem - Related Functions

Related Functions

- DosAddMuxWaitSem
- DosCloseMuxWaitSem
- DosCreateMuxWaitSem
- DosDeleteMuxWaitSem
- DosQueryMuxWaitSem
- DosWaitMuxWaitSem

DosOpenMuxWaitSem - Example Code

This example creates a MuxWait semaphore, adding two event semaphores to its record list at creation time. Then, it shows how to open it. Some return code checking has been omitted for brevity.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX       hmuxFromCreate  = NULLHANDLE;   /* Handle returned by Create */
 HMUX       hmuxFromOpen    = NULLHANDLE;   /* Handle returned by Open */
 HEV        hev[2]          = {0};          /* Event semaphores */
 SEMRECORD  apsr[2]         = {{0}};        /* Semaphore records */
 APIRET     rc              = NO_ERROR;     /* Return code */
 ULONG      ulSem           = 0;

rc = DosCreateEventSem("\\SEM32\\E0", &hev[0], 0, FALSE);
apsr[0].hsemCur = (HSEM) hev[0],
apsr[0].ulUser = 0;

rc = DosCreateEventSem("\\SEM32\\E1", &hev[1], 0, FALSE);
apsr[1].hsemCur = (HSEM) hev[1],
apsr[1].ulUser = 0;

rc = DosCreateMuxWaitSem("\\SEM32\\MUXWAIT1", &hmuxFromCreate, 2L,
                         apsr, DCMW_WAIT_ALL);
if (rc != NO_ERROR) {
   printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
   return 1;
} else {
   printf ("DosCreateMuxWaitSem returns handle = 0x%x\n", hmuxFromCreate);
}
      /*********************************************************************/
```

```
      /* The following call would normally be done from another program. */
      /*****************************************************************/

rc = DosOpenMuxWaitSem("\\SEM32\\MUXWAIT1",
                       &hmuxFromOpen);     /* Handle returned */
if (rc != NO_ERROR) {
  printf("DosOpenMuxWaitSem error:  return code = %u\n", rc);
  return 1;
} else {
   printf ("DosOpenMuxWaitSem   returns handle = 0x%x\n", hmuxFromOpen);
}

return NO_ERROR;
}
```

DosOpenMuxWaitSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosOpenQueue

DosOpenQueue - Syntax

Gives a client process access to a queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

PPID       ppid;     /*  A pointer to the process identification of the queue's server process. */
PHQUEUE    phq;      /*  A pointer to the write handle of the queue to be opened. */
PSZ        pszName;  /*  A pointer to the ASCIIZ name of the queue to be opened. */
APIRET     ulrc;     /*  Return Code. */

ulrc = DosOpenQueue(ppid, phq, pszName);
```

DosOpenQueue Parameter - ppid

ppid (PPID) - output
A pointer to the process identification of the queue's server process.

DosOpenQueue Parameter - phq

phq (PHQUEUE) - output
A pointer to the write handle of the queue to be opened.

DosOpenQueue Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the queue to be opened.

This is the name that was specified by the server process when it created the queue with DosCreateQueue. The name string must include \QUEUES\ as the first element of the path.

DosOpenQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosOpenQueue returns one of the following values:

0	NO_ERROR
334	ERROR_QUE_NO_MEMORY
341	ERROR_QUE_PROC_NO_ACCESS
343	ERROR_QUE_NAME_NOT_EXIST

For a full list of error codes, see Errors.

DosOpenQueue - Parameters

ppid (PPID) - output
A pointer to the process identification of the queue's server process.

phq (PHQUEUE) - output
A pointer to the write handle of the queue to be opened.

pszName (PSZ) - input
A pointer to the ASCIIZ name of the queue to be opened.

This is the name that was specified by the server process when it created the queue with DosCreateQueue. The name string must include \QUEUES\ as the first element of the path.

ulrc (APIRET) - returns
Return Code.

DosOpenQueue returns one of the following values:

0	NO_ERROR
334	ERROR_QUE_NO_MEMORY

341	ERROR_QUE_PROC_NO_ACCESS
343	ERROR_QUE_NAME_NOT_EXIST

For a full list of error codes, see Errors.

DosOpenQueue - Remarks

DosOpenQueue opens a queue for a client process.

If the queue was created by a call to the 16-bit DosCreateQueue function, the queue is not accessible to 32-bit DosOpenQueue requests, and ERROR_QUE_PROC_NO_ACCESS is returned.

DosOpenQueue - Related Functions

Related Functions

- DosCloseQueue
- DosCreateQueue
- DosPeekQueue
- DosPurgeQueue
- DosQueryQueue
- DosReadQueue
- DosWriteQueue

DosOpenQueue - Example Code

This example creates and opens a queue named "SPECIAL.QUE", writes to it, reads from it, and then closes it.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSPROCESS  /* DOS thread and process values */
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

   PSZ         szQueueName  = "\\QUEUES\\SPECIAL.QUE";
   HQUEUE      hqSpecialQue = NULLHANDLE; /* Queue handle                   */
   PSZ         DataBuffer   = "";         /* Data buffer for queue data     */
   REQUESTDATA Request      = {0};        /* Reques */
   PID         pidOwner     = 0;
   ULONG       ulDataLen    = 0;          /* Length of data returned        */
   BYTE        ElemPrty     = 0;          /* Priority of element (returned) */
   APIRET      rc           = NO_ERROR;   /* Return code                    */

   rc = DosCreateQueue(&hqSpecialQue,    /* Queue handle                   */
            QUE_FIFO |                   /* First-In First-Out order       */
            QUE_CONVERT_ADDRESS,         /* Convert 16-bit addresses to 32 */
            szQueueName);                /* Name of the queue to create    */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosOpenQueue (&pidOwner,         /* PID of queue owner             */
                      &hqSpecialQue,     /* Handle for created queue       */
                      szQueueName);      /* Name of the queue to open      */
   if (rc!= NO_ERROR) {
      printf ("DosOpenQueue error: return code = %u\n", rc);
      return 1;
```

```
   }
   DataBuffer = "To be, or not to be.  That is the question...";
   rc = DosWriteQueue (hqSpecialQue,        /* Queue to write to           */
                       12345L,              /* Request data                */
                       strlen(DataBuffer),  /* Length of data to write     */
                       DataBuffer,          /* Pointer to data             */
                       0L);              /* Priority (not applicable here) */
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "";                  /* Clear the DataBuffer */
   Request.pid = pidOwner;           /* process ID for the DosReadQueue */

   rc = DosReadQueue (hqSpecialQue,          /* Queue to read from          */
                      &Request,              /* Request data from write     */
                      &ulDataLen,            /* Length of data returned     */
                      (PVOID) &DataBuffer,   /* The data                    */
                      0L,                    /* Remove first element        */
                      DCWW_WAIT,             /* Wait for available data     */
                      &ElemPrty,             /* Priority of data (returned) */
                      0L);             /* Semaphore to use when not waiting */

   if (rc!= NO_ERROR) {
      printf ("DosReadQueue error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosReadQueue returns: '%s'\n", DataBuffer);
      printf ("      (Request data = %u)\n", Request.ulData);
   }

   rc = DosCloseQueue(hqSpecialQue);         /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosOpenQueue - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosOpenVDD

DosOpenVDD - Syntax

Opens a virtual device driver (VDD), and returns a handle for it.

```
#define INCL_DOSMVDM
#include <os2.h>

PSZ       pszVDD;  /*  The ASCIIZ name of the virtual device driver to be opened. */
PHVDD     phvdd;   /*  A pointer to the HVDD in which the handle of the virtual device driver is returned. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosOpenVDD(pszVDD, phvdd);
```

DosOpenVDD Parameter - pszVDD

pszVDD (PSZ) - input
The ASCIIZ name of the virtual device driver to be opened.

DosOpenVDD Parameter - phvdd

phvdd (PHVDD) - output
A pointer to the HVDD in which the handle of the virtual device driver is returned.

DosOpenVDD Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosOpenVDD returns one of the following values:

0	NO_ERROR
643	ERROR_VDD_NOT_FOUND
644	ERROR_INVALID_CALLER

For a full list of error codes, see Errors.

DosOpenVDD - Parameters

pszVDD (PSZ) - input
The ASCIIZ name of the virtual device driver to be opened.

phvdd (PHVDD) - output
A pointer to the HVDD in which the handle of the virtual device driver is returned.

ulrc (APIRET) - returns
Return Code.

DosOpenVDD returns one of the following values:

0	NO_ERROR
643	ERROR_VDD_NOT_FOUND

644 ERROR_INVALID_CALLER

For a full list of error codes, see Errors.

DosOpenVDD - Remarks

DosOpenVDD opens a virtual device driver, and returns a handle for it.

If *pszVDD* specifies the name of an OS/2 virtual device driver, the returned handle allows an OS/2 protected-mode application to communicate with a virtual device driver by issuing DosRequestVDD.

Issue DosCloseVDD to close the handle of the virtual device driver.

DosOpenVDD - Related Functions

Related Functions

- DosCloseVDD
- DosOpenVDD
- DosRequestVDD

DosOpenVDD - Example Code

The following is NOT a complete C program. It is simply intended to provide an idea of how a protected-mode OS/2 process can communicate with a virtual device driver (VDD).

This example shows a protected-mode process calling a hypothetical VDD with a request to read a string of bytes from the VDD. Assume that the session identifier of the specified DOS session has been placed into SessionID already and that the sample virtual device driver has registered the name "VDD007" with the operating system.

```
#define INCL_DOSMVDM     /* Multiple DOS sessions values */
#define INCL_DOSERRORS   /* DOS Error values */
#include <os2.h>
#include <stdio.h>

UCHAR   VDDName[10] = "VDD007";    /* Name of VDD */
HVDD    VDDHandle   = NULLHANDLE;  /* Handle of VDD */
SGID    SessionID   = 0;        /* Session identifier (should be initialized */
ULONG   Command     = 3;        /* VDD function code (hypothetical) */
APIRET  rc          = NO_ERROR;    /* Return code */
UCHAR   InputBuffer[30]  = "Your command here";  /* Command information    */
UCHAR   OutputBuffer[30] = "";                   /* Output data (returned) */

   rc = DosOpenVDD(VDDName, &VDDHandle);
   if (rc != NO_ERROR) {
      printf("VDD %s was not found.\n", rc);
      return 1;
   }

   rc = DosRequestVDD(VDDHandle,              /* Handle of VDD */
                      SessionID,              /* Session ID */
                      Command,                /* Command to send to VDD */
                      sizeof(InputBuffer),    /* Length of input */
                      InputBuffer,            /* Input buffer */
                      sizeof(OutputBuffer),   /* Length of output area */
                      OutputBuffer);          /* Output from command */
   if (rc != NO_ERROR) {
       printf("DosRequestVDD error: return code = %u\n", rc);
       return 1;
   }
```

```
    rc = DosCloseVDD(VDDHandle);               /* Close the VDD */
    if (rc != NO_ERROR) {
        printf("DosCloseVDD error: return code = %u\n", rc);
        return 1;
    }
```

DosOpenVDD - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPeekNPipe

DosPeekNPipe - Syntax

Examines the data in a named pipe without removing it.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE         hpipe;      /*  The named-pipe handle to examine. */
PVOID         pBuf;       /*  A pointer to the output buffer. */
ULONG         cbBuf;      /*  The number of bytes to be read. */
PULONG        pcbActual;  /*  A pointer a ULONG in which the number of bytes that were read is returned. */
PAVAILDATA    pAvail;     /*  A pointer to the AVAILDATA in which the number of bytes that were available is ret
PULONG        pState;     /*  A pointer to the ULONG in which the state of the named pipe is returned. */
APIRET        ulrc;       /*  Return Code. */

ulrc = DosPeekNPipe(hpipe, pBuf, cbBuf, pcbActual,
         pAvail, pState);
```

DosPeekNPipe Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to examine.

DosCreateNPipe returns the server handle; DosOpen returns the client handle.

DosPeekNPipe Parameter - pBuf

pBuf (PVOID) - output
A pointer to the output buffer.

DosPeekNPipe Parameter - cbBuf

cbBuf (ULONG) - input
The number of bytes to be read.

DosPeekNPipe Parameter - pcbActual

pcbActual (PULONG) - output
A pointer a ULONG in which the number of bytes that were read is returned.

DosPeekNPipe Parameter - pAvail

pAvail (PAVAILDATA) - output
A pointer to the AVAILDATA in which the number of bytes that were available is returned.

DosPeekNPipe Parameter - pState

pState (PULONG) - output
A pointer to the ULONG in which the state of the named pipe is returned.

Possible values are shown in the following list:

1	NP_STATE_DISCONNECTED The pipe is in a *disconnected state* immediately after a call to DosCreateNPipe, or DosDisConnectNPipe. A disconnected pipe cannot accept a call to DosOpen. The server must issue DosDisConnectNPipe before the pipe can be opened by a client.
2	NP_STATE_LISTENING The pipe is in a *listening state* after the server issues DosConnectNPipe. A listening pipe is ready to accept a DosOpen request. If the pipe is not in a listening state, DosOpen returns ERROR_PIPE_BUSY.
3	NP_STATE_CONNECTED The pipe is in a *connected state* after a client has successfully issued DosOpen. The connected pipe allows the server and the client to read and write to the pipe, provided both have valid handles.
4	NP_STATE_CLOSING The pipe is in a *closing state* after the last DosClose request has been made to the pipe by either the client or the

server. When DosClose has been issued for the client handle and all of its duplicates, the client end of the pipe is closed. The serving end must acknowledge the closing of the client end by issuing either DosDisConnectNPipe or DosClose. Issuing DosClose reallocates the pipe.

DosPeekNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosPeekNPipe returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosPeekNPipe - Parameters

hpipe (HPIPE) - input
The named-pipe handle to examine.

DosCreateNPipe returns the server handle; DosOpen returns the client handle.

pBuf (PVOID) - output
A pointer to the output buffer.

cbBuf (ULONG) - input
The number of bytes to be read.

pcbActual (PULONG) - output
A pointer a ULONG in which the number of bytes that were read is returned.

pAvail (PAVAILDATA) - output
A pointer to the AVAILDATA in which the number of bytes that were available is returned.

pState (PULONG) - output
A pointer to the ULONG in which the state of the named pipe is returned.

Possible values are shown in the following list:

1 NP_STATE_DISCONNECTED

The pipe is in a *disconnected state* immediately after a call to DosCreateNPipe, or DosDisConnectNPipe. A disconnected pipe cannot accept a call to DosOpen. The server must issue DosDisConnectNPipe before the pipe can be opened by a client.

2 NP_STATE_LISTENING

The pipe is in a *listening state* after the server issues DosConnectNPipe. A listening pipe is ready to accept a DosOpen request. If the pipe is not in a listening state, DosOpen returns ERROR_PIPE_BUSY.

3 NP_STATE_CONNECTED

The pipe is in a *connected state* after a client has successfully issued DosOpen. The connected pipe allows the server and the client to read and write to the pipe, provided both have valid handles.

4 NP_STATE_CLOSING

The pipe is in a *closing state* after the last DosClose request has been made to the pipe by either the client or the

server. When DosClose has been issued for the client handle and all of its duplicates, the client end of the pipe is closed. The serving end must acknowledge the closing of the client end by issuing either DosDisConnectNPipe or DosClose. Issuing DosClose reallocates the pipe.

ulrc (APIRET) - returns
Return Code.

DosPeekNPipe returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosPeekNPipe - Remarks

DosPeekNPipe examines the current contents of a named pipe without removing it. It also returns information about the state of the pipe.

DosPeekNPipe never blocks, even if the pipe is in blocking mode; if the pipe cannot be accessed immediately, ERROR_PIPE_BUSY is returned. Because this function does not block, it returns only what is currently in the pipe. Thus, if a message pipe is being examined, only a portion of a message may be returned, even though the specified buffer length could accommodate the entire message.

The value returned in *pState* can be used by the client or the server to determine the current state of the pipe and to take appropriate action.

Clients of named pipes created with the NP_ACCESS_INBOUND access mode cannot use the DosPeekNPipe function. If the named pipe's client uses the DosPeekNPipe function, the function returns error code ERROR_ACCESS_DENIED.

DosPeekNPipe - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosPeekNPipe - Example Code

This example handles the client side of a pipe. It opens an existing named pipe, queries the pipe handle type and pipe state, sends a message to the host, reads the host reply, and finally closes the pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   PIPEINFO PipeBuffer[4]         = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   ULONG    PipeState             = 0;
   ULONG    HandType              = 0;
   ULONG    FlagWord              = 0;
   ULONG    BytesRead             = 0;

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to pipe.\n");
   rc = DosQueryHType(PipeHandle, &HandType, &FlagWord);
   if (rc != NO_ERROR) {
      printf("DosQueryHType error: error code = %u\n", rc);
      return 1;
   }  else printf("Handle type value is %u\n", HandType);

   rc = DosPeekNPipe(PipeHandle, Buffer, sizeof(Buffer),
                     &BytesRead, &BytesAvail, &PipeState);
   if (rc != NO_ERROR) {
      printf("DosPeekNPipe error: error code = %u\n", rc);
      return 1;
   }  else printf("Pipe status value is %u\n\n", PipeState);

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);   /* Flush above printf out to display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }

   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST: %s\n\n", message);

   rc = DosClose(PipeHandle);
   /* Should check if (rc != NO_ERROR) here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosPeekNPipe - Topics

Select an item:
Syntax

Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPeekQueue

DosPeekQueue - Syntax

Examines a queue element without removing it from the queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

HQUEUE          hq;         /*  The handle of the queue from which an element is to be examined. */
PREQUESTDATA    pRequest;   /*   A pointer to the REQUESTDATA in which PID and event codes are returned. */
PULONG          pcbData;    /*  A pointer to the length of the examined data. */
PPVOID          ppbuf;      /*  A pointer to the address of the element that is to be examined. */
PULONG          element;    /*  A pointer to an indicator that specifies whether to start at the beginning of th
BOOL32          nowait;     /*  The action to be performed when there are no elements in the queue. */
PBYTE           ppriority;  /*  A pointer to the element's priority value. */
HEV             hsem;       /*  The handle of an event semaphore that is to be posted when the data is added to
APIRET          ulrc;       /*  Return Code. */

ulrc = DosPeekQueue(hq, pRequest, pcbData,
         ppbuf, element, nowait, ppriority,
         hsem);
```

DosPeekQueue Parameter - hq

hq (HQUEUE) - input
The handle of the queue from which an element is to be examined.

DosPeekQueue Parameter - pRequest

pRequest (PREQUESTDATA) - output
A pointer to the REQUESTDATA in which PID and event codes are returned.

DosPeekQueue Parameter - pcbData

pcbData (PULONG) - output
A pointer to the length of the examined data.

This field is the same as the *pcbData* that was furnished by DosWriteQueue when the element was added to the queue.

DosPeekQueue Parameter - ppbuf

ppbuf (PPVOID) - output
A pointer to the address of the element that is to be examined.

This field may or may not be the same as the *ppbuf* that was returned by DosWriteQueue when the element was added to the queue. If QUE_CONVERT_ADDRESS was specified when the queue was created, then the addresses of any elements that are written to the queue by the 16-bit DosWriteQueue function are converted to 32-bit addresses.

DosPeekQueue Parameter - element

element (PULONG) - in/out
A pointer to an indicator that specifies whether to start at the beginning of the queue or at a particular element.

Possible values are described in the following list:

0	Set by the application to indicate "examine the first element in the queue," according to the order that was specified when the queue was created (FIFO, LIFO, or priority).
Any Other Value	Set by the DosPeekQueue function to identify the element that has been examined (output), or by the owner to indicate "examine the next element" (input).

Note: By contrast, when a DosReadQueue request follows DosPeekQueue, DosReadQueue removes the same element that is identified by *element*, not the next element in the queue.

DosPeekQueue Parameter - nowait

nowait (BOOL32) - input
The action to be performed when there are no elements in the queue.

Possible values are shown in the following list:

0	DCWW_WAIT The requesting thread waits until an element is placed in the queue.
1	DCWW_NOWAIT The requesting thread does not wait, and DosPeekQueue returns with ERROR_QUE_EMPTY.

DosPeekQueue Parameter - ppriority

ppriority (PBYTE) - output
A pointer to the element's priority value.

This is the value that was specified for *ppriority* when DosWriteQueue added the element to the queue. *ppriority* is a numeric value in the range of 0 to 15, with 15 being the highest priority.

DosPeekQueue Parameter - hsem

hsem (HEV) - input
The handle of an event semaphore that is to be posted when the data is added to the queue and *nowait* is set to 1.

The semaphore may be either private or shared, depending on whether the queue is shared across processes.

hsem is ignored if *nowait* is set to 0.

Note: The first time an event-semaphore handle is supplied in a DosPeekQueue or DosReadQueue request for which *nowait* is set to 1, the handle is saved by the system. The same handle must be supplied in all subsequent DosPeekQueue and DosReadQueue requests that are issued for that queue.

DosPeekQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosPeekQueue returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
330	ERROR_QUE_PROC_NOT_OWNED
333	ERROR_QUE_ELEMENT_NOT_EXIST
337	ERROR_QUE_INVALID_HANDLE
340	ERROR_QUE_PREV_AT_END
342	ERROR_QUE_EMPTY
433	ERROR_QUE_INVALID_WAIT

For a full list of error codes, see Errors.

DosPeekQueue - Parameters

hq (HQUEUE) - input
The handle of the queue from which an element is to be examined.

pRequest (PREQUESTDATA) - output
A pointer to the REQUESTDATA in which PID and event codes are returned.

pcbData (PULONG) - output
A pointer to the length of the examined data.

This field is the same as the *pcbData* that was furnished by DosWriteQueue when the element was added to the queue.

ppbuf (PPVOID) - output
A pointer to the address of the element that is to be examined.

This field may or may not be the same as the *ppbuf* that was returned by DosWriteQueue when the element was added to the queue. If QUE_CONVERT_ADDRESS was specified when the queue was created, then the addresses of any elements that are written to the

queue by the 16-bit DosWriteQueue function are converted to 32-bit addresses.

element (PULONG) - in/out
A pointer to an indicator that specifies whether to start at the beginning of the queue or at a particular element.

Possible values are described in the following list:

0	Set by the application to indicate "examine the first element in the queue," according to the order that was specified when the queue was created (FIFO, LIFO, or priority).
Any Other Value	Set by the DosPeekQueue function to identify the element that has been examined (output), or by the owner to indicate "examine the next element" (input).

Note: By contrast, when a DosReadQueue request follows DosPeekQueue, DosReadQueue removes the same element that is identified by *element*, not the next element in the queue.

nowait (BOOL32) - input
The action to be performed when there are no elements in the queue.

Possible values are shown in the following list:

0	DCWW_WAIT The requesting thread waits until an element is placed in the queue.
1	DCWW_NOWAIT The requesting thread does not wait, and DosPeekQueue returns with ERROR_QUE_EMPTY.

ppriority (PBYTE) - output
A pointer to the element's priority value.

This is the value that was specified for *ppriority* when DosWriteQueue added the element to the queue. *ppriority* is a numeric value in the range of 0 to 15, with 15 being the highest priority.

hsem (HEV) - input
The handle of an event semaphore that is to be posted when the data is added to the queue and *nowait* is set to 1.

The semaphore may be either private or shared, depending on whether the queue is shared across processes.

hsem is ignored if *nowait* is set to 0.

Note: The first time an event-semaphore handle is supplied in a DosPeekQueue or DosReadQueue request for which *nowait* is set to 1, the handle is saved by the system. The same handle must be supplied in all subsequent DosPeekQueue and DosReadQueue requests that are issued for that queue.

ulrc (APIRET) - returns
Return Code.

DosPeekQueue returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
330	ERROR_QUE_PROC_NOT_OWNED
333	ERROR_QUE_ELEMENT_NOT_EXIST
337	ERROR_QUE_INVALID_HANDLE
340	ERROR_QUE_PREV_AT_END
342	ERROR_QUE_EMPTY
433	ERROR_QUE_INVALID_WAIT

For a full list of error codes, see Errors.

DosPeekQueue - Remarks

DosPeekQueue examines a queue element without removing it from the queue. This function can be used only by the queue's server process and its threads.

If the *nowait* parameter is set to 1, an event semaphore must be provided so that the calling thread can determine when an element has been placed into the queue. The semaphore is created by calling DosCreateEventSem, and its handle is supplied in the *hsem* parameter of DosPeekQueue.

The first time an event-semaphore handle is supplied in a DosPeekQueue or DosReadQueue request for which *nowait* has been set to 1, the handle is saved by the system. The same handle must be supplied in all subsequent DosPeekQueue and DosReadQueue requests that are issued for the same queue; if a different handle is supplied, ERROR_INVALID_PARAMETER is returned.

When a client process adds an element to the queue, the system automatically opens and posts the semaphore. The server can either issue DosQueryEventSem periodically to determine whether the semaphore has been posted, or it can issue DosWaitEventSem. DosWaitEventSem causes the calling thread to block until the semaphore is posted.

After the event semaphore has been posted, the calling thread must call DosPeekQueue again to examine the newly added queue element.

DosPeekQueue - Related Functions

Related Functions

- DosCloseQueue
- DosCreateQueue
- DosOpenQueue
- DosPurgeQueue
- DosQueryQueue
- DosReadQueue
- DosWriteQueue

DosPeekQueue - Example Code

This example creates and opens a named queue. It writes a message to it, peeks the message, and finally closes it.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSPROCESS  /* DOS thread and process values */
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

   PSZ         szQueueName  = "\\QUEUES\\AnyQueueNameHere";
   HQUEUE      hqAnyQueue   = NULLHANDLE; /* Queue handle                    */
   PSZ         DataBuffer   = "";         /* Data buffer for queue data      */
   REQUESTDATA Request      = {0};        /* Request                         */
   ULONG       ulDataLen    = 0,          /* Length of data returned         */
               ulElemCode   = 0;          /* Element code (input/output)     */
   BYTE        ElemPrty     = 0;          /* Priority of element (returned)  */
   APIRET      rc           = NO_ERROR;   /* Return code                     */
   rc = DosCreateQueue(&hqAnyQueue,      /* Queue handle                    */
            QUE_FIFO |                   /* First-In First-Out order        */
            QUE_CONVERT_ADDRESS,         /* Convert 16-bit addresses to 32  */
            szQueueName);                /* Name of the queue to create     */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "Start of the data... Middle of data...  Data ends.";
   rc = DosWriteQueue (hqAnyQueue,          /* Queue to write to           */
                       87654321L,           /* Request data                */
                       strlen(DataBuffer),  /* Length of data to write     */
                       DataBuffer,          /* Pointer to data             */
                       0L);              /* Priority (not applicable here) */
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "";                         /* Clear the DataBuffer        */
   rc = DosPeekQueue (hqAnyQueue,           /* Handle of queue             */
                      &Request,             /* Request data for element    */
                      &ulDataLen,           /* Length of data returned     */
                      (PVOID) &DataBuffer,  /* Data returned               */
```

```
                      &ulElemCode,           /* Input: 0 reads next element
                                                Output:  peeked element id  */
                      DCWW_WAIT,             /* Wait for data               */
                      &ElemPrty,             /* Priority of element         */
                      0L);                   /* Semaphore (not used here)   */
  if (rc != NO_ERROR) {
     printf ("DosPeekQueue error : return code = %u\n", rc);
     return 1;
  } else {
     printf ("DosPeekQueue returns: '%s'\n", DataBuffer);
     printf ("  (Request data = %u)\n", Request.ulData);
  }

  rc = DosCloseQueue(hqAnyQueue);         /* Close the queue */
  if (rc!= NO_ERROR) {
     printf ("DosCloseQueue error: return code = %u\n", rc);
     return 1;
  }

  return NO_ERROR;
}
```

DosPeekQueue - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPerfSysCall

DosPerfSysCall - Syntax

Retrieve system performance information.

```
#define INCL_BASE
#include <os2.h>

ULONG     ulCommand;  /*  Command */
ULONG     ulParm1;    /*  Command specific parameter 1 */
ULONG     ulParm2;    /*  Command specific parameter 2 */
ULONG     ulParm3;    /*  Command specific parameter 3 */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosPerfSysCall(ulCommand, ulParm1,
         ulParm2, ulParm3);
```

DosPerfSysCall Parameter - ulCommand

ulCommand (ULONG) - input
Command

DosPerfSysCall accepts the following command:

CMD_KI_RDCNT (0x63)

Read CPU utilization information in both uniprocessor and symmetric multi-processor (SMP) environments by taking a snapshot of the time stamp counters. To determine CPU utilization, the application must compute the difference between two time stamp snapshots using 64-bit arithmetic. See the example code for details.

DosPerfSysCall Parameter - ulParm1

ulParm1 (ULONG) - in/out
Command specific parameter 1

Parameter usage depends on the command specified:

CMD_KI_RDCNT

Pointer to an array of CPUUTIL structures, with one entry for each processor. On a 4-way SMP machine, the array definition would be:

```
CPUUTIL CPUUtil[4];   /* For processor IDs 0, 1, 2, and 3 */
```

since processor numbers and arrays are both 0-based. *ulParm1* would be set to the address of CPUUtil[0].

ulParm2 and *ulParm3* are not used and should be set to zero.

DosPerfSysCall Parameter - ulParm2

ulParm2 (ULONG) - in/out
Command specific parameter 2

Parameter usage depends on the command specified:

CMD_KI_RDCNT

0 (reserved)

DosPerfSysCall Parameter - ulParm3

ulParm3 (ULONG) - in/out
Command specific parameter 3

Parameter usage depends on the command specified:

CMD_KI_RDCNT

0 (reserved)

DosPerfSysCall Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosPerfSysCall returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION

For a full list of error codes, see Errors.

DosPerfSysCall - Parameters

ulCommand (ULONG) - input
Command

DosPerfSysCall accepts the following command:

CMD_KI_RDCNT (0x63)
Read CPU utilization information in both uniprocessor and symmetric multi-processor (SMP) environments by taking a snapshot of the time stamp counters. To determine CPU utilization, the application must compute the difference between two time stamp snapshots using 64-bit arithmetic. See the example code for details.

ulParm1 (ULONG) - in/out
Command specific parameter 1

Parameter usage depends on the command specified:

CMD_KI_RDCNT
Pointer to an array of CPUUTIL structures, with one entry for each processor. On a 4-way SMP machine, the array definition would be:

```
CPUUTIL CPUUtil[4];   /* For processor IDs 0, 1, 2, and 3 */
```

since processor numbers and arrays are both 0-based. *ulParm1* would be set to the address of CPUUtil[0].

ulParm2 and *ulParm3* are not used and should be set to zero.

ulParm2 (ULONG) - in/out
Command specific parameter 2

Parameter usage depends on the command specified:

CMD_KI_RDCNT
0 (reserved)

ulParm3 (ULONG) - in/out
Command specific parameter 3

Parameter usage depends on the command specified:

CMD_KI_RDCNT
0 (reserved)

ulrc (APIRET) - returns
Return Code.

DosPerfSysCall returns one of the following values:

0	NO_ERROR

1 ERROR_INVALID_FUNCTION

For a full list of error codes, see Errors.

DosPerfSysCall - Remarks

DosPerfSysCall is a general purpose performance function. This function accepts four parameters. The first parameter is the command requested. The other three parameters are command specific. DosPerfSysCall is intended to be extended by IBM in the future for other performance related functions.

Some functions of DosPerfSysCall may have a dependency on Intel Pentium or Pentium-Pro support. If a function cannot be provided because OS/2 is not running on an processor with the required features, a return code will indicate an attempt to use an unsupported function.

The **perfutil.h** file referenced in the example code may or may not be shipped as part of the Toolkit. In the event that it is not, its contents are:

```
#ifdef __cplusplus
  extern "C" {
#endif

#ifndef  PERFCALL_INCLUDED
#define  PERFCALL_INCLUDED

/*
   DosPerfSysCall Function Prototype
*/

/* The  ordinal for DosPerfSysCall (in BSEORD.H) */
/* is defined as ORD_DOS32PERFSYSCALL         */

APIRET APIENTRY DosPerfSysCall(ULONG ulCommand, ULONG ulParm1, ULONG ulParm2, ULONG ulParm3);

/***
 *
 * CPU Utilization
 * ---------------
 *
 **/

#define   CMD_KI_RDCNT    (0x63)

 typedef struct _CPUUTIL {
   ULONG ulTimeLow;     /* Low 32 bits of time stamp      */
   ULONG ulTimeHigh;    /* High 32 bits of time stamp     */
   ULONG ulIdleLow;     /* Low 32 bits of idle time       */
   ULONG ulIdleHigh;    /* High 32 bits of idle time      */
   ULONG ulBusyLow;     /* Low 32 bits of busy time       */
   ULONG ulBusyHigh;    /* High 32 bits of busy time      */
   ULONG ulIntrLow;     /* Low 32 bits of interrupt time  */
   ULONG ulIntrHigh;    /* High 32 bits of interrupt time */
  } CPUUTIL;

 typedef CPUUTIL *PCPUUTIL;

#ifdef __cplusplus
  }
#endif
```

It may also be necessary to define the ordinal for this function:

```
#define ORD_DOS32PERFSYSCALL            976
```

DosPerfSysCall - Related Functions

Related Functions

- DosQuerySysInfo

DosPerfSysCall - Example Code

This example uses DosPerfSysCall to obtain CPU Utilization information on a uniprocessor.

```
#define INCL_BASE

#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <perfutil.h>

/*
   Convert 8-byte (low, high) time value to double
*/
#define LL2F(high, low) (4294967296.0*(high)+(low))


/* This is a 1 processor example */

void main (int argc, char *argv[])
{
    APIRET      rc;
    int         i, iter, sleep_sec;
    double      ts_val, ts_val_prev;
    double      idle_val, idle_val_prev;
    double      busy_val, busy_val_prev;
    double      intr_val, intr_val_prev;
    CPUUTIL     CPUUtil;

    if ((argc < 2) || (*argv[1] < '1') || (*argv[1] > '9')) {
        fprintf(stderr, "usage: %s [1-9]\n", argv[0]);
        exit(0);
    }
    sleep_sec = *argv[1] - '0';

    iter = 0;
    do {
        rc = DosPerfSysCall(CMD_KI_RDCNT,(ULONG) &CPUUtil,0,0);
        if (rc) {
            fprintf(stderr, "CMD_KI_RDCNT failed rc = %d\n",rc);
            exit(1);
        }
        ts_val = LL2F(CPUUtil.ulTimeHigh, CPUUtil.ulTimeLow);
        idle_val = LL2F(CPUUtil.ulIdleHigh, CPUUtil.ulIdleLow);
        busy_val = LL2F(CPUUtil.ulBusyHigh, CPUUtil.ulBusyLow);
        intr_val = LL2F(CPUUtil.ulIntrHigh, CPUUtil.ulIntrLow);

        if (iter > 0) {
            double  ts_delta = ts_val - ts_val_prev;
            printf("idle: %4.2f%%  busy: %4.2f%%  intr: %4.2f%%\n",
                   (idle_val - idle_val_prev)/ts_delta*100.0,
                   (busy_val - busy_val_prev)/ts_delta*100.0,
                   (intr_val - intr_val_prev)/ts_delta*100.0);
        }

        ts_val_prev = ts_val;
        idle_val_prev = idle_val;
        busy_val_prev = busy_val;
        intr_val_prev = intr_val;

        iter++;
        DosSleep(1000*sleep_sec);

    } while (1);
}
```

DosPerfSysCall - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPhysicalDisk

DosPhysicalDisk - Syntax

Obtains information about partitionable disks.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     function;  /*  The type of information to obtain about the partitionable disks. */
PVOID     pBuf;      /*  The address of the buffer where the returned information is placed. */
ULONG     cbBuf;     /*  The length, in bytes, of the data buffer. */
PVOID     pParams;   /*  The address of the buffer used for input parameters. */
ULONG     cbParams;  /*  The length, in bytes, of the parameter buffer. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosPhysicalDisk(function, pBuf, cbBuf,
         pParams, cbParams);
```

DosPhysicalDisk Parameter - function

function (ULONG) - input

The type of information to obtain about the partitionable disks.

Possible values are shown in the following list:

1 — INFO_COUNT_PARTITIONABLE_DISKS
Obtain the total number of partitionable disks.

2 — INFO_GETIOCTLHANDLE
Obtain a handle to use with Category 09h Physical Disk Control IOCtl Commands.

3 — INFO_FREEIOCTLHANDLE
Release a handle for a partitionable disk.

DosPhysicalDisk Parameter - pBuf

pBuf (PVOID) - input
The address of the buffer where the returned information is placed.

The output data for each function is shown in the following list (all lengths are in bytes):

Function	DataLen	Returned Information
1	2	Total number of partitionable disks in the system (1-based).
2	2	Handle for the specified partitionable disk for the Category 09h Physical Disk Control IOCtl Commands.
3	0	None - pointer must be zero.

DosPhysicalDisk Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of the data buffer.

DosPhysicalDisk Parameter - pParams

pParams (PVOID) - input
The address of the buffer used for input parameters.

The input parameters required for each function are as follows (all lengths are in bytes):

Function	ParmLen	Input Parameters
1	0	None - must be set to zero.
2	string length	ASCIIZ string that specifies the partitionable disk.
3	2	Handle obtained from Function 2.

The ASCIIZ string used to specify the partitionable disk must be of the following format:

```
number : <null byte>
```

Where:

number	specifies the partitionable disk number (1-based) in ASCII.
colon (:)	must be present.
<null byte>	is the byte of zero for the ASCIIZ string.

DosPhysicalDisk Parameter - cbParams

cbParams (ULONG) - input
The length, in bytes, of the parameter buffer.

DosPhysicalDisk Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosPhysicalDisk returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosPhysicalDisk - Parameters

function (ULONG) - input
The type of information to obtain about the partitionable disks.

Possible values are shown in the following list:

1 INFO_COUNT_PARTITIONABLE_DISKS
Obtain the total number of partitionable disks.

2 INFO_GETIOCTLHANDLE
Obtain a handle to use with Category 09h Physical Disk Control IOCtl Commands.

3 INFO_FREEIOCTLHANDLE
Release a handle for a partitionable disk.

pBuf (PVOID) - input
The address of the buffer where the returned information is placed.

The output data for each function is shown in the following list (all lengths are in bytes):

Function	DataLen	Returned Information
1	2	Total number of partitionable disks in the system (1-based).
2	2	Handle for the specified partitionable disk for the Category 09h Physical Disk Control IOCtl Commands.
3	0	None - pointer must be zero.

cbBuf (ULONG) - input
The length, in bytes, of the data buffer.

pParams (PVOID) - input
The address of the buffer used for input parameters.

The input parameters required for each function are as follows (all lengths are in bytes):

Function	ParmLen	Input Parameters
1	0	None - must be set to zero.
2	string length	ASCIIZ string that specifies the partitionable disk.
3	2	Handle obtained from Function 2.

The ASCIIZ string used to specify the partitionable disk must be of the following format:

```
number : <null byte>
```

Where:

number	specifies the partitionable disk number (1-based) in ASCII.
colon (:)	must be present.
<null byte>	is the byte of zero for the ASCIIZ string.

cbParams (ULONG) - input
The length, in bytes, of the parameter buffer.

ulrc (APIRET) - returns
Return Code.

DosPhysicalDisk returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosPhysicalDisk - Remarks

DosPhysicalDisk obtains information about partitionable disks. The handle returned for the specified partitionable disk can *only* be used with the DosDevIOCtl function for the Category 09h Physical Disk Control IOCtl Commands. Use of the handle for a physical partitionable disk is not permitted for handle-based file system functions, such as DosRead or DosClose.

DosPhysicalDisk - Related Functions

Related Functions

- DosBeep
- DosDevConfig
- DosDevIOCtl

DosPhysicalDisk - Example Code

This example obtains the total number of partitionable disks in the system. A partitionable disk is a physical disk drive that can be formatted into partitions.

```
#define INCL_DOSDEVICES   /* Device values    */
#define INCL_DOSERRORS    /* DOS error values */
#include <os2.h>
#include <stdio.h>
```

```
int main (VOID) {

 USHORT  usNumDrives  = 0;                  /* Data return buffer        */
 ULONG   ulDataLen    = sizeof(USHORT);     /* Data return buffer length */
 APIRET  rc           = NO_ERROR;           /* Return code               */

 /* Request a count of the number of partitionable disks in the system */

    rc = DosPhysicalDisk(INFO_COUNT_PARTITIONABLE_DISKS,
                         &usNumDrives,
                         ulDataLen,
                         NULL,         /* No parameter for this function */
                         0L);

    if (rc != NO_ERROR) {
       printf("DosPhysicalDisk error: return code = %u\n", rc);
       return 1;
    } else {
       printf("DosPhysicalDisk:  %u partitionable disk(s)\n",usNumDrives);
    }

   return NO_ERROR;
}
```

DosPhysicalDisk - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPostEventSem

DosPostEventSem - Syntax

Posts an event semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HEV       hev;   /*  The handle of the event semaphore to post. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosPostEventSem(hev);
```

DosPostEventSem Parameter - hev

hev (HEV) - input
The handle of the event semaphore to post.

DosPostEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosPostEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
298	ERROR_TOO_MANY_POSTS
299	ERROR_ALREADY_POSTED

For a full list of error codes, see Errors.

DosPostEventSem - Parameters

hev (HEV) - input
The handle of the event semaphore to post.

ulrc (APIRET) - returns
Return Code.

DosPostEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
298	ERROR_TOO_MANY_POSTS
299	ERROR_ALREADY_POSTED

For a full list of error codes, see Errors.

DosPostEventSem - Remarks

DosPostEventSem posts an event semaphore, causing all of the threads that were blocked on DosWaitEventSem requests for that semaphore to execute.

This function can be called by any thread in the process that created the semaphore. Other processes can also call this function, but they must first gain access to the semaphore by calling DosOpenEventSem.

DosPostEventSem - Related Functions

Related Functions

- DosCloseEventSem

- DosCreateEventSem
- DosOpenEventSem
- DosQueryEventSem
- DosResetEventSem
- DosWaitEventSem

DosPostEventSem - Example Code

This example shoes how to post an event semaphore with this API.

```
 #define INCL_DOSSEMAPHORES   /* Semaphore values */
 #define INCL_DOSERRORS       /* DOS error values */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 PSZ     szSemName  = "\\SEM32\\MYTOOL\\EVENTSEM\\1"; /* Semaphore name    */
 HEV     hevEvent      = 0;                  /* Event semaphore handle    */
 APIRET  rc            = NO_ERROR;           /* Return code               */

    rc = DosCreateEventSem(szSemName,      /* Name of semaphore to create  */
                           &hevEvent,      /* Handle of semaphore returned */
                           DC_SEM_SHARED,  /* Shared semaphore             */
                           FALSE);         /* Don't want it POSTed yet     */

    if (rc != NO_ERROR) {
        printf("DosCreateEventSem error: return code = %u\n", rc);
        return 1;       }

          /* OOPS... we changed our mind, POST the semaphore */

    rc = DosPostEventSem(hevEvent);

    if (rc != NO_ERROR) {
        printf("DosPostEventSem error: return code = %u\n", rc);
        return 1;
    }

return NO_ERROR;
}
```

DosPostEventSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectClose

DosProtectClose - Syntax

Closes a handle to a file, pipe, or device.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;               /*  The handle returned by a previous call to a DosProtectOpen. */
FHLOCK    fhFileHandleLockID;  /*  The filehandle lockid obtained from DosProtectOpen. */
APIRET    ulrc;                /*  Return Code. */

ulrc = DosProtectClose(hFile, fhFileHandleLockID);
```

DosProtectClose Parameter - hFile

hFile (HFILE) - input
The handle returned by a previous call to a DosProtectOpen.

DosProtectClose Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

DosProtectClose Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectClose returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosProtectClose - Parameters

hFile (HFILE) - input
The handle returned by a previous call to a DosProtectOpen.

fhFileHandleLockID (FHLOCK) - input

The filehandle lockid obtained from DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectClose returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosProtectClose - Remarks

Issuing DosProtectClose with the handle to a file closes a handle to a file, pipe, or device.

Closing a device handle causes the device to be notified of the close, if appropriate.

Named-Pipe Considerations

DosProtectClose closes a named pipe by handle. When all handles that refer to one end of a pipe are closed, the pipe is considered broken.

If the client end closes, no other process can reopen the pipe until the serving end issues DosDisConnectNPipe, followed by DosConnectNPipe.

If the server end closes when the pipe is already broken, the pipe is immediately deallocated; otherwise, it is not deallocated until the last client handle is closed.

DosProtectClose - Related Functions

Related Functions

- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosProtectOpen
- DosResetBuffer

DosProtectClose - Example Code

This example opens or creates and opens a file named "DOSPROT.DAT", writes to it, reads from it, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;
   ULONG  ulAction       = 0;
   ULONG  ulBytesRead    = 0;
   ULONG  ulWrote        = 0;
   ULONG  ulLocal        = 0;
   UCHAR  uchFileName[20]  = "dosprot.dat",
          uchFileData[100] = " ";
```

```
   FHLOCK FileHandleLock = 0;        /* File handle lock   */
   APIRET rc             = NO_ERROR; /* Return code */

   /* Open the file dosprot.dat.  Make it read/write, open it */
   /* if it already exists and create it if it is new.     */
   rc = DosProtectOpen(uchFileName,             /* File path name          */
               &hfFileHandle,                   /* File handle             */
               &ulAction,                       /* Action taken            */
               100L,                            /* File primary allocation */
               FILE_ARCHIVED | FILE_NORMAL,     /* File attribute          */
               OPEN_ACTION_CREATE_IF_NEW |
               OPEN_ACTION_OPEN_IF_EXISTS,      /* Open function type      */
               OPEN_FLAGS_NOINHERIT |
               OPEN_SHARE_DENYNONE  |
               OPEN_ACCESS_READWRITE,           /* Open mode of the file   */
               0L,                              /* No extended attribute   */
               &FileHandleLock);                /* File handle lock id     */
   if (rc != NO_ERROR) {
      printf("DosProtectOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosProtectOpen: Action taken = %u\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n3...\n2...\n1\n");
   rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
                  (PVOID) uchFileData,       /* String to be written         */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote,                  /* Bytes actually written       */
                  FileHandleLock);           /* File handle lock id          */
   if (rc != NO_ERROR) {
      printf("DosProtectWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosProtectWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosProtectSetFilePtr (hfFileHandle,    /* File Handle          */
                     0L,                       /* Offset               */
                     FILE_BEGIN,               /* Move from BOF        */
                     &ulLocal,                 /* New location address */
                     FileHandleLock);          /* File handle lock id  */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosProtectRead (hfFileHandle,         /* File Handle                 */
                uchFileData,                  /* String to be read           */
                100L,                         /* Length of string to be read */
                &ulBytesRead,                 /* Bytes actually read         */
                FileHandleLock);              /* File handle lock id         */
   if (rc != NO_ERROR) {
      printf("DosProtectRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf("DosProtectRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
   if (rc != NO_ERROR) {
      printf("DosProtectClose error: return code = %u\n", rc);
      return 1;
    }
   return NO_ERROR;
}
```

DosProtectClose - Topics

Select an item:
Syntax

Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectEnumAttribute

DosProtectEnumAttribute - Syntax

Identifies names and lengths of extended attributes for a specific file or subdirectory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     ulRefType;           /*  A value that indicates whether pvFile points to a handle or to an ASCIIZ name
PVOID     pvFile;              /*  The address of the file handle, or the file name. */
ULONG     ulEntry;             /*  Ordinal of an entry in the file object's EA list, which indicates where in th
PVOID     pvBuf;               /*  Address of the buffer where EA information is returned. */
ULONG     cbBuf;               /*  The length, in bytes, of the buffer pointed to by pvBuf. */
PULONG    pulCount;            /*  A pointer to a ULONG containing the number of EAs. */
ULONG     ulInfoLevel;         /*  Level of information required. */
FHLOCK    fhFileHandleLockID;  /*  The filehandle lockid returned from DosProtectOpen. */
APIRET    ulrc;                /*  Return Code. */

ulrc = DosProtectEnumAttribute(ulRefType,
         pvFile, ulEntry, pvBuf, cbBuf, pulCount,
         ulInfoLevel, fhFileHandleLockID);
```

DosProtectEnumAttribute Parameter - ulRefType

ulRefType (ULONG) - input
A value that indicates whether *pvFile* points to a handle or to an ASCIIZ name.

Possible values are shown in the following list:

0	ENUMEA_REFTYPE_FHANDLE Handle of a file.
1	ENUMEA_REFTYPE_PATH ASCIIZ name of a file or subdirectory.

DosProtectEnumAttribute Parameter - pvFile

pvFile (PVOID) - input
The address of the file handle, or the file name.

If *ulRefType* equals ENUMEA_REFTYPE_FHANDLE, this field contains the address of the handle of a file returned by DosOpen. If *ulRefType* equals ENUMEA_REFTYPE_PATH, this field contains the ASCIIZ name of a file or subdirectory.

DosProtectEnumAttribute Parameter - ulEntry

ulEntry (ULONG) - input
Ordinal of an entry in the file object's EA list, which indicates where in the list to begin the return of EA information.

The value 0 is reserved. A value of 1 indicates the file object's first EA; a value of 2, the second; and so on.

DosProtectEnumAttribute Parameter - pvBuf

pvBuf (PVOID) - output
Address of the buffer where EA information is returned.

Level 1 information (*ulInfoLevel* == ENUMEA_LEVEL_NO_VALUE) is returned in a data structure of type DENA2.

DosProtectEnumAttribute Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of the buffer pointed to by *pvBuf*.

DosProtectEnumAttribute Parameter - pulCount

pulCount (PULONG) - in/out
A pointer to a ULONG containing the number of EAs.

Input	Contains the number of EAs for which information is requested. A value of -1 requests that information be returned for as many EAs whose information fits in *pvBuf*.
Output	Contains the actual number of EAs for which information is returned. When this value is greater than 1, enumerated information is returned in a series of DENA2 data structures. Each data structure is aligned on a doubleword boundary. The first field of the data structure (*oNextEntryOffset*) contains the number of bytes to the beginning of the next data structure. A value of zero for *oNextEntryOffset* indicates that this is the last data structure.

DosProtectEnumAttribute Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
Level of information required.

Only the value 1 (ENUMEA_LEVEL_NO_VALUE) can be specified, indicating return of level 1 information.

DosProtectEnumAttribute Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned from DosProtectOpen.

Required only when *ulRefType* is equal to zero. Otherwise, value is ignored.

DosProtectEnumAttribute Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectEnumAttribute returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosProtectEnumAttribute - Parameters

ulRefType (ULONG) - input
A value that indicates whether *pvFile* points to a handle or to an ASCIIZ name.

Possible values are shown in the following list:

0	ENUMEA_REFTYPE_FHANDLE Handle of a file.
1	ENUMEA_REFTYPE_PATH ASCIIZ name of a file or subdirectory.

pvFile (PVOID) - input
The address of the file handle, or the file name.

If *ulRefType* equals ENUMEA_REFTYPE_FHANDLE, this field contains the address of the handle of a file returned by DosOpen. If *ulRefType* equals ENUMEA_REFTYPE_PATH, this field contains the ASCIIZ name of a file or subdirectory.

ulEntry (ULONG) - input
Ordinal of an entry in the file object's EA list, which indicates where in the list to begin the return of EA information.

The value 0 is reserved. A value of 1 indicates the file object's first EA; a value of 2, the second; and so on.

pvBuf (PVOID) - output
Address of the buffer where EA information is returned.

Level 1 information (*ulInfoLevel* == ENUMEA_LEVEL_NO_VALUE) is returned in a data structure of type DENA2.

cbBuf (ULONG) - input
The length, in bytes, of the buffer pointed to by *pvBuf*.

pulCount (PULONG) - in/out
A pointer to a ULONG containing the number of EAs.

Input	Contains the number of EAs for which information is requested. A value of -1 requests that information be returned for as many EAs whose information fits in *pvBuf*.
Output	Contains the actual number of EAs for which information is returned. When this value is greater than 1, enumerated information is returned in a series of DENA2 data structures. Each data structure is aligned on a doubleword boundary. The first field of the data structure (*oNextEntryOffset*) contains the number of bytes to the beginning of the next data structure. A value of zero for *oNextEntryOffset* indicates that this is the last data structure.

ulInfoLevel (ULONG) - input
Level of information required.

Only the value 1 (ENUMEA_LEVEL_NO_VALUE) can be specified, indicating return of level 1 information.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned from DosProtectOpen.

Required only when *ulRefType* is equal to zero. Otherwise, value is ignored.

ulrc (APIRET) - returns
Return Code.

DosProtectEnumAttribute returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosProtectEnumAttribute - Remarks

The structure that DosProtectEnumAttribute returns is used to calculate the size of the buffer needed to hold the full extended attribute (FEA2) information for a DosQueryPathInfo or DosProtectQueryFileInfo call that actually gets the FEA2. The buffer size is calculated as follows:

Four bytes (for *oNextEntryOffset*) +
One byte (for *fEA*) +
One byte (for *cbName*) +
Two bytes (for *cbValue*) +
Value of *cbName* (for the name of the EA) +
One byte (for terminating NULL in *cbName*) +
Value of *cbValue* (for the value of the EA)

Each entry must start on a doubleword boundary.

A process can continue through a file's EA list by reissuing DosProtectEnumAttribute with *ulEntry* set to the value specified in the previous call, plus the value returned in *pulCount*.

DosProtectEnumAttribute does not control the specific ordering of EAs; it merely identifies them. Extended attributes can have multiple readers and writers, just as the files they are associated with can. If a file is open in a sharing mode that allows other processes to modify the file's EA list, repetitively calling DosProtectEnumAttribute to back up to an EA's position may return inconsistent results. For example with DosProtectSetFileInfo or DosSetPathInfo, another process can edit the EA list between calls by your process to DosProtectEnumAttribute.

Therefore, the EA returned when *ulEntry* is 11 for the first call might not be the same EA returned when *ulEntry* is 11 for the next call.

To prevent EAs from being modified between calls to DosProtectEnumAttribute for a specified file handle or file name, the calling function must open the file in deny-write sharing mode before it calls DosProtectEnumAttribute. If a subdirectory name is specified, modification by other processes is not a concern, because no sharing is possible.

When ENUEMEA_REFTYPE_PATH is specified for *ulRefType* , the EAs returned are current only when the call was made, and may have been changed by another thread or process since then.

DosProtectEnumAttribute - Related Functions

Related Functions

- DosCreateDir
- DosProtectOpen
- DosProtectQueryFileInfo
- DosProtectSetFileInfo
- DosQueryPathInfo
- DosSetPathInfo

DosProtectEnumAttribute - Example Code

This sample gets all the extended attributes (EAs) from a file named "ATTRIB.EXE" in the OS2 directory, and displays the names.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {
   UCHAR    EnumBuf[200] = {0};      /* Data Buffer */
   ULONG    ulEnumCnt    = 0;        /* Count of entries to return */
   FEA2     *ptr         = NULL;     /* Pointer to data items returned */
   ULONG    ulTemp       = 0;
   APIRET   rc           = NO_ERROR; /* Return code  */
   ULONG    i            = 0;        /* Loop index */
   FHLOCK FileHandleLock = 0;        /* File handle lock   */

   ulEnumCnt = (ULONG)-1; /* Request as many attributes as will fit in buffer */

   rc = DosProtectEnumAttribute(ENUMEA_REFTYPE_PATH,   /* ASCIIZ string       */
                        "c:\\os2\\attrib.exe", /* Name of file             */
                        1L,                    /* Start with first attribute */
                        (PVOID)&EnumBuf,       /* Buffer for information     */
                        sizeof(EnumBuf),
                        &ulEnumCnt,
                        ENUMEA_LEVEL_NO_VALUE,  /* Request level 1 info      */
                        FileHandleLock);       /* File handle lock           */
   if (rc != NO_ERROR) {
      printf("DosProtectEnumAttribute error: return code = %u\n", rc);
      return 1;
   }

   ptr = (FEA2 *)EnumBuf; /* Mask the buffer pointer to an FEA2 structure */

   printf ("Attributes found = %u\n", ulEnumCnt);

   for (i = 0; i < ulEnumCnt; i++) {
      printf ("name = %s\n", ptr->szName);       /* increment the ptr */
      ulTemp = ptr->oNextEntryOffset + (ULONG)ptr; /* with the value in */
      ptr = (FEA2 *)ulTemp;                      /* oNextEntryOffset  */
   } /* endfor */                              /* to access next record */

   return NO_ERROR;
}
```

DosProtectEnumAttribute - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectOpen

DosProtectOpen - Syntax

Opens a new file, an existing file, or a replacement for an existing file and returns a protected file handle. An open file can have extended attributes.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ         pszFileName;          /*  Address of the ASCIIZ path name of the file or device to be opened. */
PHFILE      phf;                  /*  Address of the handle for the file. */
PULONG      pulAction;            /*  A pointer to the ULONG in which the value that specifies the action taken b
ULONG       cbFile;               /*  New logical size of the file (end of data, EOD), in bytes. */
ULONG       ulAttribute;          /*  File attributes. */
ULONG       fsOpenFlags;          /*  The action to be taken depending on whether the file exists or does not exi
ULONG       fsOpenMode;           /*  The mode of the open function. */
PEAOP2      peaop2;               /*  A pointer to an extended attribute buffer. */
PFHLOCK     pfhFileHandleLockID;
APIRET      ulrc;                 /*  Return Code. */

ulrc = DosProtectOpen(pszFileName, phf, pulAction,
         cbFile, ulAttribute, fsOpenFlags,
         fsOpenMode, peaop2, pfhFileHandleLockID);
```

DosProtectOpen Parameter - pszFileName

pszFileName (PSZ) - input
Address of the ASCIIZ path name of the file or device to be opened.

DosProtectOpen Parameter - phf

phf (PHFILE) - output
Address of the handle for the file.

DosProtectOpen Parameter - pulAction

pulAction (PULONG) - output
A pointer to the ULONG in which the value that specifies the action taken by DosProtectOpen is returned.

If DosProtectOpen fails, this value has no meaning. Otherwise, it is one of the following values:

1	FILE_EXISTED File already existed.
2	FILE_CREATED File was created.
3	FILE_TRUNCATED File existed and was changed to a given size (file was replaced).

DosProtectOpen Parameter - cbFile

cbFile (ULONG) - input
New logical size of the file (end of data, EOD), in bytes.

This parameter is significant only when creating a new file or replacing an existing one. Otherwise, it is ignored. It is an error to create or replace a file with a nonzero length if the *fsOpenMode* Access-Mode flag is set to read-only.

DosProtectOpen Parameter - ulAttribute

ulAttribute (ULONG) - input
File attributes.

This parameter contains the following bit fields:

Bit	Description
31-6	Reserved, must be 0.
5	FILE_ARCHIVED (0x00000020) File has been archived.
4	FILE_DIRECTORY (0x00000010) File is a subdirectory.
3	Reserved, must be 0.
2	FILE_SYSTEM (0x00000004) File is a system file.
1	FILE_HIDDEN (0x00000002) File is hidden and does not appear in a directory listing.
0	FILE_READONLY (0x00000001) File can be read from, but not written to.

0	FILE_NORMAL (0x00000000) File can be read from or written to.

File attributes apply only if the file is created.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates a read-only file that has been archived.

DosProtectOpen Parameter - fsOpenFlags

fsOpenFlags (ULONG) - input
The action to be taken depending on whether the file exists or does not exist.

This parameter contains the following bit fields.:

Bits	Description
31-8	Reserved, must be 0.
7-4	The following flags apply if the file does not exist:
	0000 OPEN_ACTION_FAIL_IF_NEW Open an existing file; fail if the file does not exist.
	0001 OPEN_ACTION_CREATE_IF_NEW Create the file if the file does not exist.
3-0	The following flags apply if the file does not exist:
	0000 OPEN_ACTION_FAIL_IF_EXISTS Open the file; fail if the file already exists.
	0001 OPEN_ACTION_OPEN_IF_EXISTS Open the file if it already exists.
	0010 OPEN_ACTION_REPLACE_IF_EXISTS Replace the file if it already exists.

DosProtectOpen Parameter - fsOpenMode

fsOpenMode (ULONG) - input
The mode of the open function.

This parameter contains the following bit fields:

Bit	Description
29-16	Reserved, must be zero.
30	OPEN_FLAGS_PROTECTED_HANDLE (0x40000000) Protected file handle flag. 0 Unprotected Handle 1 Protected Handle Protected handle requires the *pfhFileHandleLockID* to be specified on subsequent DosProtectxxxx calls. Unprotected handle requires the *pfhFileHandleLockID* value to be specified as zero on subsequent DosProtectxxxx calls. An unprotected handle may be used with the unprotected calls such as DosRead and DosWrite.

31 Reserved, must be zero.

15 OPEN_FLAGS_DASD (x00008000)
Direct Open flag:

0 *pszFileName* represents a file to be opened normally.

1 *pszFileName* is "drive:" (such as C: or A:), and represents a mounted disk or diskette volume to be opened for direct access.

14 OPEN_FLAGS_WRITE_THROUGH (0x00004000)
Write-Through flag:

0 Writes to the file may go through the file-system driver's cache. The file-system driver writes the sectors when the cache is full or the file is closed.

1 Writes to the file may go through the file-system driver's cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous write operations.

This bit flag is not inherited by child processes.

13 OPEN_FLAGS_FAIL_ON_ERROR (0x00002000)
Fail-Errors flag. Media I/O errors are handled as follows:

0 Reported through the system critical-error handler.
1 Reported directly to the caller by way of a return code.

Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always get reported directly to the caller by way of return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O calls.

This flag bit is not inherited by child processes.

12 OPEN_FLAGS_NO_CACHE (0x00001000)
No-Cache/Cache flag:

0 The file-system driver should place data from I/O operations into its cache.

1 I/O operations to the file need not be done through the file-system driver's cache.

The setting of this bit determines whether file-system drivers should place data into the cache. Like the write-through bit, this is a per-handle bit, and is not inherited by child processes.

11 Reserved; must be 0.

10-8 The locality of reference flags contain information about how the application is to get access to the file. The values are as follows:

000 OPEN_FLAGS_NO_LOCALITY (0x00000000)
No locality known.

001 OPEN_FLAGS_SEQUENTIAL (0x00000100)
Mainly sequential access.

010 OPEN_FLAGS_RANDOM (0x00000200)
Mainly random access.

011 OPEN_FLAGS_RANDOMSEQUENTIAL (0x00000300)
Random with some locality.

7 OPEN_FLAGS_NOINHERIT (0x00000080)
Inheritance flag:

0 File handle is inherited by a process created from a call to DosExecPgm.

1 File handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing Mode flags. This field defines any restrictions to file access placed by the caller on other processes. The values are as follows:

	001	OPEN_SHARE_DENYREADWRITE (0x00000010) Deny read/write access.
	010	OPEN_SHARE_DENYWRITE (0x00000020) Deny write access.
	011	OPEN_SHARE_DENYREAD (0x00000030) Deny read access.
	100	OPEN_SHARE_DENYNONE (0x00000040) Deny neither read nor write access (deny none).
	Any other value is invalid.	
3	Reserved; must be 0.	
2-0	Access-Mode flags. This field defines the file access required by the caller. The values are as follows:	
	000	OPEN_ACCESS_READONLY (0x00000000) Read-only access
	001	OPEN_ACCESS_WRITEONLY (0x00000001) Write-only access
	010	OPEN_ACCESS_READWRITE (0x00000002) Read/write access.
	Any other value is invalid, as are any other combinations.	

File sharing requires the cooperation of sharing processes. This cooperation is communicated through sharing and access modes. Any sharing restrictions placed on a file opened by a process are removed when the process closes the file with a DosClose request.

Sharing Mode

Specifies the type of file access that other processes may have. For example, if other processes can continue to read the file while your process is operating on it, specify Deny Write. The sharing mode prevents other processes from writing to the file but still allows them to read it.

Access Mode

Specifies the type of file access (access mode) needed by your process. For example, if your process requires read/write access, and another process has already opened the file with a sharing mode of Deny None, your DosProtectOpen request succeeds. However, if the file is open with a sharing mode of Deny Write, the process is denied access.

If the file is inherited by a child process, all sharing and access restrictions also are inherited.

If an open file handle is duplicated by a call to DosDupHandle, all sharing and access restrictions also are duplicated.

DosProtectOpen Parameter - peaop2

peaop2 (PEAOP2) - in/out

A pointer to an extended attribute buffer.

Input	The address of the extended-attribute buffer, which contains an EAOP2 structure. The *fpFEA2List* field in the EAOP2 structure. points to a data area where the relevant FEA2 list is to be found. The *fpGEA2List* and *oError* fields are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code from DosProtectOpen is the error code for that error condition. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the file. If extended attributes are not to be defined or modified, the pointer *peaop2* must be set to zero.

DosProtectOpen Parameter - pfhFileHandleLockID

pfhFileHandleLockID (PFHLOCK) - output
The address of the 32-bit lockid for the file handle.

DosProtectOpen Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectOpen returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

For a full list of error codes, see Errors.

DosProtectOpen - Parameters

pszFileName (PSZ) - input
Address of the ASCIIZ path name of the file or device to be opened.

phf (PHFILE) - output
Address of the handle for the file.

pulAction (PULONG) - output
A pointer to the ULONG in which the value that specifies the action taken by DosProtectOpen is returned.

If DosProtectOpen fails, this value has no meaning. Otherwise, it is one of the following values:

1	FILE_EXISTED File already existed.
2	FILE_CREATED File was created.
3	FILE_TRUNCATED File existed and was changed to a given size (file was replaced).

cbFile (ULONG) - input
New logical size of the file (end of data, EOD), in bytes.

This parameter is significant only when creating a new file or replacing an existing one. Otherwise, it is ignored. It is an error to create or replace a file with a nonzero length if the *fsOpenMode* Access-Mode flag is set to read-only.

ulAttribute (ULONG) - input
File attributes.

This parameter contains the following bit fields:

Bit	Description
31-6	Reserved, must be 0.
5	FILE_ARCHIVED (0x00000020) File has been archived.
4	FILE_DIRECTORY (0x00000010) File is a subdirectory.
3	Reserved, must be 0.
2	FILE_SYSTEM (0x00000004) File is a system file.
1	FILE_HIDDEN (0x00000002) File is hidden and does not appear in a directory listing.
0	FILE_READONLY (0x00000001) File can be read from, but not written to.
0	FILE_NORMAL (0x00000000) File can be read from or written to.

File attributes apply only if the file is created.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates a read-only file that has been archived.

fsOpenFlags (ULONG) - input
The action to be taken depending on whether the file exists or does not exist.

This parameter contains the following bit fields.:

Bits	Description	
31-8	Reserved, must be 0.	
7-4	The following flags apply if the file does not exist:	
	0000	OPEN_ACTION_FAIL_IF_NEW Open an existing file; fail if the file does not exist.
	0001	OPEN_ACTION_CREATE_IF_NEW Create the file if the file does not exist.
3-0	The following flags apply if the file does not exist:	
	0000	OPEN_ACTION_FAIL_IF_EXISTS Open the file; fail if the file already exists.
	0001	OPEN_ACTION_OPEN_IF_EXISTS Open the file if it already exists.
	0010	OPEN_ACTION_REPLACE_IF_EXISTS Replace the file if it already exists.

fsOpenMode (ULONG) - input
The mode of the open function.

This parameter contains the following bit fields:

Bit	Description
29-16	Reserved, must be zero.
30	OPEN_FLAGS_PROTECTED_HANDLE (0x40000000) Protected file handle flag.

0 Unprotected Handle
1 Protected Handle

Protected handle requires the *pfhFileHandleLockID* to be specified on subsequent DosProtectxxxx calls.

Unprotected handle requires the *pfhFileHandleLockID* value to be specified as zero on subsequent DosProtectxxxx calls. An unprotected handle may be used with the unprotected calls such as DosRead and DosWrite.

31 Reserved, must be zero.

15 OPEN_FLAGS_DASD (x00008000)
Direct Open flag:

0 *pszFileName* represents a file to be opened normally.

1 *pszFileName* is "drive:" (such as C: or A:), and represents a mounted disk or diskette volume to be opened for direct access.

14 OPEN_FLAGS_WRITE_THROUGH (0x00004000)
Write-Through flag:

0 Writes to the file may go through the file-system driver's cache. The file-system driver writes the sectors when the cache is full or the file is closed.

1 Writes to the file may go through the file-system driver's cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous write operations.

This bit flag is not inherited by child processes.

13 OPEN_FLAGS_FAIL_ON_ERROR (0x00002000)
Fail-Errors flag. Media I/O errors are handled as follows:

0 Reported through the system critical-error handler.
1 Reported directly to the caller by way of a return code.

Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always get reported directly to the caller by way of return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O calls.

This flag bit is not inherited by child processes.

12 OPEN_FLAGS_NO_CACHE (0x00001000)
No-Cache/Cache flag:

0 The file-system driver should place data from I/O operations into its cache.

1 I/O operations to the file need not be done through the file-system driver's cache.

The setting of this bit determines whether file-system drivers should place data into the cache. Like the write-through bit, this is a per-handle bit, and is not inherited by child processes.

11 Reserved; must be 0.

10-8 The locality of reference flags contain information about how the application is to get access to the file. The values are as follows:

000 OPEN_FLAGS_NO_LOCALITY (0x00000000)
No locality known.

001 OPEN_FLAGS_SEQUENTIAL (0x00000100)
Mainly sequential access.

010 OPEN_FLAGS_RANDOM (0x00000200)
Mainly random access.

011 OPEN_FLAGS_RANDOMSEQUENTIAL (0x00000300)
Random with some locality.

7 OPEN_FLAGS_NOINHERIT (0x00000080)

Inheritance flag:

0 File handle is inherited by a process created from a call to DosExecPgm.

1 File handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing Mode flags. This field defines any restrictions to file access placed by the caller on other processes. The values are as follows:

001 OPEN_SHARE_DENYREADWRITE (0x00000010)
Deny read/write access.

010 OPEN_SHARE_DENYWRITE (0x00000020)
Deny write access.

011 OPEN_SHARE_DENYREAD (0x00000030)
Deny read access.

100 OPEN_SHARE_DENYNONE (0x00000040)
Deny neither read nor write access (deny none).

Any other value is invalid.

3 Reserved; must be 0.

2-0 Access-Mode flags. This field defines the file access required by the caller. The values are as follows:

000 OPEN_ACCESS_READONLY (0x00000000)
Read-only access

001 OPEN_ACCESS_WRITEONLY (0x00000001)
Write-only access

010 OPEN_ACCESS_READWRITE (0x00000002)
Read/write access.

Any other value is invalid, as are any other combinations.

File sharing requires the cooperation of sharing processes. This cooperation is communicated through sharing and access modes. Any sharing restrictions placed on a file opened by a process are removed when the process closes the file with a DosClose request.

Sharing Mode

Specifies the type of file access that other processes may have. For example, if other processes can continue to read the file while your process is operating on it, specify Deny Write. The sharing mode prevents other processes from writing to the file but still allows them to read it.

Access Mode

Specifies the type of file access (access mode) needed by your process. For example, if your process requires read/write access, and another process has already opened the file with a sharing mode of Deny None, your DosProtectOpen request succeeds. However, if the file is open with a sharing mode of Deny Write, the process is denied access.

If the file is inherited by a child process, all sharing and access restrictions also are inherited.

If an open file handle is duplicated by a call to DosDupHandle, all sharing and access restrictions also are duplicated.

peaop2 (PEAOP2) - in/out
A pointer to an extended attribute buffer.

Input The address of the extended-attribute buffer, which contains an EAOP2 structure. The *fpFEA2List* field in the EAOP2 structure. points to a data area where the relevant FEA2 list is to be found. The *fpGEA2List* and *oError* fields are ignored.

Output *fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code from DosProtectOpen is the error code for that error condition. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the file. If extended attributes are not to be defined or modified, the pointer *peaop2* must be set to zero.

pfhFileHandleLockID (PFHLOCK) - output
The address of the 32-bit lockid for the file handle.

ulrc (APIRET) - returns
Return Code.

DosProtectOpen returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

For a full list of error codes, see Errors.

DosProtectOpen - Remarks

A successful DosProtectOpen request returns a handle and a 32-bit lockid for accessing the file. The read/write pointer is set at the first byte of the file. The position of the pointer can be changed with DosProtectSetFilePtr or by read and write operations on the file.

The file's date and time can be queried with DosProtectQueryFileInfo. They are set with DosProtectSetFileInfo.

The read-only attribute of a file can be set with the ATTRIB command.

ulAttribute cannot be set to Volume Label. To set volume-label information, issue DosProtectSetFileInfo with a logical drive number. Volume labels cannot be opened.

cbFile affects the size of the file only when the file is new or is a replacement. If an existing file is opened, *cbFile* is ignored. To change the size of the existing file, issue DosProtectSetFileSize.

The value in *cbFile* is a recommended size. If the full size cannot be allocated, the open request may still succeed. The file system makes a reasonable attempt to allocate the new size in an area that is as nearly contiguous as possible on the medium. When the file size is extended, the values of the new bytes are undefined.

The Direct Open bit provides direct access to an entire disk or diskette volume, independent of the file system. This mode of opening the volume that is currently on the drive returns a handle to the calling function; the handle represents the logical volume as a single file. The calling function specifies this handle with a DosDevIOCtl Category 8, DSK_LOCKDRIVE request to prevent other processes from accessing the logical volume. When you are finished using the logical volume, issue a DosDevIOCtl Category 8, DSK_UNLOCKDRIVE request to allow other processes to access the logical volume.

The file-handle state bits can be set by DosProtectOpen and DosProtectSetFHState. An application can query the file-handle state bits, as well as the rest of the Open Mode field, by issuing DosProtectQueryFHState.

You can use an EAOP2 structure to set extended attributes in *peaop2* when creating a file, replacing an existing file, or truncating an existing file. No extended attributes are set when an existing file is just opened.

A replacement operation is logically equivalent to atomically deleting and re-creating the file. This means that any extended attributes associated with the file also are deleted before the file is re-created.

The *pfhFileHandleLockID* returned is required on each of the DosProtectxxxxx functions. An incorrect *pfhFileHandleLockID* on subsequent DosProtectxxx calls results in an ERROR_ACCESS_DENIED return code.

The DosProtectxxxxx functions can be used with a NULL filehandle lockid, if the subject filehandle was obtained from DosOpen.

DosProtectOpen - Related Functions

Related Functions

- DosDevIOCtl
- DosDupHandle
- DosProtectClose
- DosProtectSetFileInfo
- DosProtectSetFilePtr
- DosProtectSetFileSize
- DosQueryHType
- DosSetMaxFH
- DosSetRelMaxFH

DosProtectOpen - Example Code

This example opens or creates and opens a file named "DOSPROT.DAT", writes to it, reads from it, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;
   ULONG  ulAction       = 0;
   ULONG  ulBytesRead    = 0;
   ULONG  ulWrote        = 0;
   ULONG  ulLocal        = 0;
   UCHAR  uchFileName[20]  = "dosprot.dat",
          uchFileData[100] = " ";
   FHLOCK FileHandleLock = 0;        /* File handle lock   */
   APIRET rc             = NO_ERROR; /* Return code */

   /* Open the file dosprot.dat.  Make it read/write, open it */
   /* if it already exists and create it if it is new.     */
   rc = DosProtectOpen(uchFileName,             /* File path name          */
               &hfFileHandle,                   /* File handle             */
               &ulAction,                       /* Action taken            */
               100L,                            /* File primary allocation */
               FILE_ARCHIVED | FILE_NORMAL,     /* File attribute          */
               OPEN_ACTION_CREATE_IF_NEW |
               OPEN_ACTION_OPEN_IF_EXISTS,      /* Open function type      */
               OPEN_FLAGS_NOINHERIT |
               OPEN_SHARE_DENYNONE  |
               OPEN_ACCESS_READWRITE,           /* Open mode of the file   */
               0L,                              /* No extended attribute   */
               &FileHandleLock);                /* File handle lock id     */
   if (rc != NO_ERROR) {
      printf("DosProtectOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosProtectOpen: Action taken = %u\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n3...\n2...\n1\n");

   rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
                  (PVOID) uchFileData,       /* String to be written         */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote,                  /* Bytes actually written       */
                  FileHandleLock);           /* File handle lock id          */
   if (rc != NO_ERROR) {
      printf("DosProtectWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosProtectWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosProtectSetFilePtr (hfFileHandle,    /* File Handle          */
```

```
                     0L,                     /* Offset               */
                     FILE_BEGIN,             /* Move from BOF        */
                     &ulLocal,               /* New location address */
                     FileHandleLock);        /* File handle lock id  */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosProtectRead (hfFileHandle,         /* File Handle                 */
                uchFileData,                  /* String to be read           */
                100L,                         /* Length of string to be read */
                &ulBytesRead,                 /* Bytes actually read         */
                FileHandleLock);              /* File handle lock id         */
   if (rc != NO_ERROR) {
      printf("DosProtectRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf("DosProtectRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
   if (rc != NO_ERROR) {
      printf("DosProtectClose error: return code = %u\n", rc);
      return 1;
    }
   return NO_ERROR;
}
```

DosProtectOpen - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectQueryFHState

DosProtectQueryFHState - Syntax

Queries the state of the specified protected file handle.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;              /*  Handle of the file to be queried. */
PULONG    pMode;              /*  Pointer to the ULONG in which the content of the fsOpenMode field defined in
FHLOCK    fhFileHandleLockID; /*  The lock id of the protected file handle. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectQueryFHState(hFile, pMode,
         fhFileHandleLockID);
```

DosProtectQueryFHState Parameter - hFile

hFile (HFILE) - input
Handle of the file to be queried.

DosProtectQueryFHState Parameter - pMode

pMode (PULONG) - output
Pointer to the ULONG in which the content of the *fsOpenMode* field defined in a previous DosOpen is returned.

Possible modes include those in the following list:

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag: 0 — *pszFileName* from a DosOpen function represents a file to be opened normally. 1 — *pszFileName* is "drive:" (such as C: or A:). It represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Write operations to the file go through the file system buffer cache. 1 — Write operations to the file may go through the file system buffer cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit is set to 1 because the data must be written to the medium for synchronous write operations. The Write-Through flag bit is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always are reported directly to the caller by a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. The Fail-Errors flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00001000) Cache or No-Cache: 0 — The disk driver should place data from I/O operations into the cache on this file. 1 — I/O operations to the file need not be done through the disk-driver cache. The setting of this bit determines whether it is worth caching the data for file-systems drivers and device drivers. This bit, like the Write-Through bit, is a per-handle bit. This bit is not inherited by child processes.

11-8 Reserved bits.

7 OPEN_FLAGS_NOINHERIT (0x00000080)
Inheritance flag:

0 The file handle is inherited by a process that is created by issuing DosExecPgm.

1 The file handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing-Mode flags: Define the operations other processes can perform on the file:

001 OPEN_SHARE_DENYREADWRITE
Deny read and write access.

010 OPEN_SHARE_DENYWRITE
Deny write access.

011 OPEN_SHARE_DENYREAD
Deny read access.

100 OPEN_SHARE_DENYNONE
Deny neither read nor write access (deny none).

Any other value is invalid.

3 Reserved.

2-0 Access-Mode flags. File access is assigned as follows:

000 OPEN_ACCESS_READONLY
Read-only access.

001 OPEN_ACCESS_WRITEONLY
Write-only access.

010 OPEN_ACCESS_READWRITE
Read and write access.

Any other value is invalid.

DosProtectQueryFHState Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The lock id of the protected file handle.

DosProtectQueryFHState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectQueryFHState returns one of the following values:

0 NO_ERROR
6 ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosProtectQueryFHState - Parameters

hFile (HFILE) - input
Handle of the file to be queried.

pMode (PULONG) - output
Pointer to the ULONG in which the content of the *fsOpenMode* field defined in a previous DosOpen is returned.

Possible modes include those in the following list:

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag: 0 — *pszFileName* from a DosOpen function represents a file to be opened normally. 1 — *pszFileName* is "drive:" (such as C: or A:). It represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Write operations to the file go through the file system buffer cache. 1 — Write operations to the file may go through the file system buffer cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit is set to 1 because the data must be written to the medium for synchronous write operations. The Write-Through flag bit is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always are reported directly to the caller by a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. The Fail-Errors flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00001000) Cache or No-Cache: 0 — The disk driver should place data from I/O operations into the cache on this file. 1 — I/O operations to the file need not be done through the disk-driver cache. The setting of this bit determines whether it is worth caching the data for file-systems drivers and device drivers. This bit, like the Write-Through bit, is a per-handle bit. This bit is not inherited by child processes.
11-8	Reserved bits.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 — The file handle is inherited by a process that is created by issuing DosExecPgm. 1 — The file handle is private to the current process. This bit is not inherited by child processes.
6-4	Sharing-Mode flags: Define the operations other processes can perform on the file:

	001	OPEN_SHARE_DENYREADWRITE Deny read and write access.
	010	OPEN_SHARE_DENYWRITE Deny write access.
	011	OPEN_SHARE_DENYREAD Deny read access.
	100	OPEN_SHARE_DENYNONE Deny neither read nor write access (deny none).
	Any other value is invalid.	
3	Reserved.	
2-0	Access-Mode flags. File access is assigned as follows:	
	000	OPEN_ACCESS_READONLY Read-only access.
	001	OPEN_ACCESS_WRITEONLY Write-only access.
	010	OPEN_ACCESS_READWRITE Read and write access.
	Any other value is invalid.	

fhFileHandleLockID (FHLOCK) - input
The lock id of the protected file handle.

ulrc (APIRET) - returns
Return Code.

DosProtectQueryFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosProtectQueryFHState - Remarks

When the application cannot handle a critical error that occurs, critical-error handling can be reset to the system. This is done by having DosSetFHState turn off the fail/errors bit and then reissuing the I/O function. The expected critical error recurs, and control is passed to the system critical-error handler. The precise time that the effect of this function is visible at the application level is unpredictable when asynchronous I/O operations are pending.

The Direct Open bit parameter is the "Direct I/O flag." It provides an access mechanism to a disk or diskette volume independent of the file system. This mode should be used only by system programs and not by application programs.

Named-Pipe Considerations

As defined by the operating system, D = 0. Other bits are as defined by DosCreateNPipe (serving end), DosOpen (client end), or the last DosSetFHState.

DosProtectQueryFHState - Related Functions

Related Functions

- DosDevIOCtl
- DosOpen
- DosSetFHState

DosProtectQueryFHState - Example Code

This example opens or creates and opens a file named "DOSPQFH.DAT" in protected mode, and queries the state of its handle using this API.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 UCHAR       uchFileName[]   = "DOSPQFH.DAT";   /* File to manipulate        */
 HFILE       fhQryFile       = 0;               /* File handle from DosOpen  */
 FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
 ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpen */
 ULONG       FHState         = 0;                 /* File Handle State       */
 APIRET      rc              = NO_ERROR;          /* Return code             */
 FHLOCK      FileHandleLock  = 0;                 /* File handle lock        */

  rc = DosProtectOpen(uchFileName, &fhQryFile,
               &ulOpenAction, 10L, FILE_NORMAL,
               OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
               OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L,
               &FileHandleLock);
  if (rc != NO_ERROR) {
     printf("DosProtectOpen error: return code = %u\n", rc);
     return 1;
  }

  rc = DosProtectQueryFHState(fhQryFile, &FHState, FileHandleLock);
  if (rc != NO_ERROR) {
      printf("DosProtectQueryFHState error: return code = %u\n", rc);
      return 1;
  } else printf("FHState is: %x\n", FHState);

    /*   Change state to indicate that data should not be cached */

    FHState &= 0x7F88;                  /* Turn off non-participating bits */
    rc = DosProtectSetFHState(fhQryFile, FHState | OPEN_FLAGS_NO_CACHE,
                              FileHandleLock);
    if (rc != NO_ERROR) {
        printf("DosProtectSetFHState error: return code = %u\n", rc);
        return 1;
    }

    rc = DosProtectClose(fhQryFile, FileHandleLock);
    /* Should check if (rc != NO_ERROR) here */

    rc = DosDelete(uchFileName);      /* Delete the file */
    if (rc != NO_ERROR) {
        printf("DosDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    } /* endif */

   return NO_ERROR;
}
```

DosProtectQueryFHState - Topics

Select an item:
Syntax
Parameters
Returns

Remarks
Example Code
Related Functions
Glossary

DosProtectQueryFileInfo

DosProtectQueryFileInfo - Syntax

Gets file information.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hf;                 /*  File handle. */
ULONG     ulInfoLevel;        /*  Level of file information required. */
PVOID     pInfo;              /*  Address of the storage area where the system returns the requested level of f
ULONG     cbInfoBuf;          /*  The length, in bytes, of pInfo. */
FHLOCK    fhFileHandleLockID; /*  The filehandle lockid returned by a previous DosProtectOpen. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectQueryFileInfo(hf, ulInfoLevel,
         pInfo, cbInfoBuf, fhFileHandleLockID);
```

DosProtectQueryFileInfo Parameter - hf

hf (HFILE) - input
File handle.

DosProtectQueryFileInfo Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
Level of file information required.

A value of 1, 2, or 3 can be specified, as shown in the following list:

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information
3	FIL_QUERYEASFROMLIST Level 3 file information

Level 4 is reserved.

The structures described in *pInfo* indicate the information returned for each of these levels.

DosProtectQueryFileInfo Parameter - pInfo

pInfo (PVOID) - output
Address of the storage area where the system returns the requested level of file information.

File information, where applicable, is at least as accurate as the most recent DosProtectClose, DosResetBuffer, DosProtectSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfo contains the FILESTATUS3 data structure, to which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfo contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an ULONG. On output, this field contains the size, in bytes, of the file's entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file's entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

Input	*pInfo* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 list defining the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be zero. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfo* is unchanged. The buffer pointed to by *fpFEA2List* is filled in with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size of the entire EA set on disk for the file, even though only a subset of attributes was requested.

DosProtectQueryFileInfo Parameter - cbInfoBuf

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfo* .

DosProtectQueryFileInfo Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned by a previous DosProtectOpen.

DosProtectQueryFileInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectQueryFileInfo returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosProtectQueryFileInfo - Parameters

hf (HFILE) - input
File handle.

ulInfoLevel (ULONG) - input
Level of file information required.

A value of 1, 2, or 3 can be specified, as shown in the following list:

1 FIL_STANDARD
Level 1 file information

2 FIL_QUERYEASIZE
Level 2 file information

3 FIL_QUERYEASFROMLIST
Level 3 file information

Level 4 is reserved.

The structures described in *pInfo* indicate the information returned for each of these levels.

pInfo (PVOID) - output
Address of the storage area where the system returns the requested level of file information.

File information, where applicable, is at least as accurate as the most recent DosProtectClose, DosResetBuffer, DosProtectSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfo contains the FILESTATUS3 data structure, to which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfo contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an ULONG. On output, this field contains the size, in bytes, of the file's entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file's entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

Input *pInfo* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 list defining the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be zero. *fpFEA2List* points to a data area where the relevant FEA2 list is

	returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfo* is unchanged. The buffer pointed to by *fpFEA2List* is filled in with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size of the entire EA set on disk for the file, even though only a subset of attributes was requested.

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfo* .

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned by a previous DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectQueryFileInfo returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosProtectQueryFileInfo - Remarks

In the FAT file system, only date and time information contained in level-1 file information can be modified. Zero is returned for the creation and access dates and times.

To return information contained in any of the file information levels, DosProtectQueryFileInfo must be able to read the open file. DosProtectQueryFileInfo works only when the file is opened for read access, with a deny-write sharing mode specified for access by other processes. If another process that has specified conflicting sharing and access modes has already opened the file, any call to DosProtectOpen will fail.

DosProtectEnumAttribute returns the lengths of extended attributes. This information can be used to calculate what size *pInfo* needs to be to hold full-extended-attribute (FEA) information returned by DosProtectQueryFileInfo when Level 3 is specified. The size of the buffer is calculated as follows:

Four bytes (for *oNextEntryOffset*) +
One byte (for *fEA*) +
One byte (for *cbName*) +
Two bytes (for *cbValue*) +
Value of *cbName* (for the name of the EA) +
One byte (for terminating NULL in *cbName*) +
Value of *cbValue* (for the value of the EA)

DosProtectQueryFileInfo - Related Functions

Related Functions

- DosProtectClose
- DosProtectOpen
- DosProtectEnumAttribute
- DosProtectSetFileInfo
- DosQueryPathInfo
- DosResetBuffer

- DosProtectSetFileSize
- DosSetPathInfo

DosProtectQueryFileInfo - Example Code

This example creates a read-only file named "DOSFDEL.DAT", then changes the file attributes to normal, and uses DosForceDelete to delete the file so that it can not be restored using UNDELETE.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 UCHAR       uchFileName[]   = "DOSFDEL.DAT";   /* File we want to delete    */
 HFILE       fhDelFile       = 0;               /* File handle from DosOpen  */
 FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
 ULONG       ulBufferSize    = sizeof(FILESTATUS3); /* File info buffer size */
 ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpen */
 APIRET      rc              = NO_ERROR;          /* Return code             */
 FHLOCK      FHLock          = 0;                 /* File handle lock        */

                /* Create a read-only file */

  rc = DosProtectOpen(uchFileName, &fhDelFile,
              &ulOpenAction, 10L, FILE_READONLY,
              OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
              OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L, &FHLock);
  if (rc != NO_ERROR) {
     printf("DosProtectOpen error: return code = %u\n", rc);
     return 1;
  }

  rc = DosProtectQueryFileInfo(fhDelFile, FIL_STANDARD,
              &fsts3FileInfo, ulBufferSize, FHLock);   /* Get standard info */
  if (rc != NO_ERROR) {
      printf("DosProtectQueryFileInfo error: return code = %u\n", rc);
      return 1;
  } else { printf("File %s created read-only.\n",uchFileName); }

    /*   Change the file attributes to be "normal"  */

    fsts3FileInfo.attrFile  = FILE_NORMAL;
    rc = DosProtectSetFileInfo(fhDelFile, FIL_STANDARD,
                        &fsts3FileInfo, ulBufferSize, FHLock);
    if (rc != NO_ERROR) {
        printf("DosProtectSetFileInfo error: return code = %u\n", rc);
        return 1;
    }

    rc = DosProtectClose(fhDelFile, FHLock);
    /* Should verify that (rc != NO_ERROR) here... */

           /* Delete the file */

    rc = DosForceDelete(uchFileName);
    if (rc != NO_ERROR) {
        printf("DosForceDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    }  /* endif */

   return NO_ERROR;
}
```

DosProtectQueryFileInfo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectRead

DosProtectRead - Syntax

Reads the specified number of bytes from a file, pipe, or device to a buffer location.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;              /*  File handle obtained from DosOpen. */
PVOID     pBuffer;            /*  Address of the buffer to receive the bytes read. */
ULONG     cbRead;             /*  The length, in bytes, of pBuffer. */
PULONG    pcbActual;          /*  Pointer to the ULONG in which the number of bytes actually read is returned.
FHLOCK    fhFileHandleLockID; /*  The filehandle lockid obtained from DosProtectOpen. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectRead(hFile, pBuffer, cbRead,
         pcbActual, fhFileHandleLockID);
```

DosProtectRead Parameter - hFile

hFile (HFILE) - input
File handle obtained from DosOpen.

DosProtectRead Parameter - pBuffer

pBuffer (PVOID) - output
Address of the buffer to receive the bytes read.

DosProtectRead Parameter - cbRead

cbRead (ULONG) - input
The length, in bytes, of *pBuffer* .

This is the number of bytes to be read.

DosProtectRead Parameter - pcbActual

pcbActual (PULONG) - output
Pointer to the ULONG in which the number of bytes actually read is returned.

DosProtectRead Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

DosProtectRead Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectRead returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE
234	ERROR_MORE_DATA

For a full list of error codes, see Errors.

DosProtectRead - Parameters

hFile (HFILE) - input
File handle obtained from DosOpen.

pBuffer (PVOID) - output
Address of the buffer to receive the bytes read.

cbRead (ULONG) - input
The length, in bytes, of *pBuffer* .

This is the number of bytes to be read.

pcbActual (PULONG) - output
Pointer to the ULONG in which the number of bytes actually read is returned.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectRead returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE
234	ERROR_MORE_DATA

For a full list of error codes, see Errors.

DosProtectRead - Remarks

The requested number of bytes might not be read. If the value returned in *pBuffer* is zero, the process tried to read from the end of the file.

A value of zero for *cbRead* is not considered an error. In such a case, the system treats the request as a null operation.

The file pointer is moved to the desired position by reading data, writing data, or issuing DosProtectSetFilePtr.

If you issue DosProtectOpen with the Direct Open flag set to 1 in the *fsOpenMode* parameter, you have direct access to an entire disk or diskette volume, independent of the file system. You must lock the logical volume before accessing it, and you must unlock the logical volume when you are finished accessing it. Issue DosDevIOCtl for Category 8, DSK_LOCKDRIVE to lock the logical volume, and for Category 8, DSK_UNLOCKDRIVE to unlock the logical volume. While the logical volume is locked, no other process can access it.

Named-Pipe Considerations

A named pipe is read as one of the following:

- A byte pipe in byte-read mode
- A message pipe in message-read mode
- A message pipe in byte-read mode.

A byte pipe must be in byte-read mode to be read; an error is returned if it is in message-read mode. All currently available data, up to the size requested, is returned.

A message pipe can be read in either message-read mode or byte-read mode. When the message pipe is in message-read mode, a read operation that is larger than the next available message returns only that message. *pcbActual* is set to indicate the size of the message returned.

A read operation that is smaller than the next available message returns with the number of bytes requested and an ERROR_MORE_DATA return code. When the reading of a message is resumed after ERROR_MORE_DATA is returned, a read operation always blocks until the next piece (or the rest) of the message can be transferred. DosPeekNPipe can be issued to determine how many bytes are left in the message.

A message pipe in byte-read mode is read as if it were a byte stream, and DosProtectRead skips over message headers. This is like reading a byte pipe in byte-read mode.

When blocking mode is set for a named pipe, a read operation blocks until data is available. In this case, the read operation never returns with *pcbActual* equal to zero, except at the end of the file. When the mode is set to message-read, messages are always read in their entirety, except when the message is bigger than the size of the read operation.

pcbActual can equal zero in nonblocking mode, but only when no data is available at the time of the read operation.

DosProtectRead - Related Functions

Related Functions

- DosProtectOpen
- DosProtectSetFilePtr
- DosProtectWrite

DosProtectRead - Example Code

This example opens or creates and opens a file named "DOSPROT.DAT". It writes a string to the file, positions the file pointer back to BOF, reads the string written, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;
   ULONG  ulAction       = 0;
   ULONG  ulBytesRead    = 0;
   ULONG  ulWrote        = 0;
   ULONG  ulLocal        = 0;
   UCHAR  uchFileName[20]  = "dosprot.dat",
          uchFileData[100] = " ";
   FHLOCK FileHandleLock = 0;        /* File handle lock   */
   APIRET rc             = NO_ERROR; /* Return code */

   /* Open the file test.dat.  Make it read/write, open it */
   /* if it already exists and create it if it is new.     */
   rc = DosProtectOpen(uchFileName,             /* File path name          */
               &hfFileHandle,                   /* File handle             */
               &ulAction,                       /* Action taken            */
               100L,                            /* File primary allocation */
               FILE_ARCHIVED | FILE_NORMAL,     /* File attribute          */
               OPEN_ACTION_CREATE_IF_NEW |
               OPEN_ACTION_OPEN_IF_EXISTS,      /* Open function type      */
               OPEN_FLAGS_NOINHERIT |
               OPEN_SHARE_DENYNONE  |
               OPEN_ACCESS_READWRITE,           /* Open mode of the file   */
               0L,                              /* No extended attribute   */
               &FileHandleLock);                /* File handle lock id     */
   if (rc != NO_ERROR) {
      printf("DosProtectOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosProtectOpen: Action taken = %u\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n3...\n2...\n1\n");
   rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
                  (PVOID) uchFileData,       /* String to be written         */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote,                  /* Bytes actually written       */
                  FileHandleLock);           /* File handle lock id          */
   if (rc != NO_ERROR) {
      printf("DosProtectWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosProtectWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosProtectSetFilePtr (hfFileHandle,    /* File Handle          */
                      0L,                      /* Offset               */
                      FILE_BEGIN,              /* Move from BOF        */
                      &ulLocal,                /* New location address */
                      FileHandleLock);         /* File handle lock id  */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosProtectRead (hfFileHandle,         /* File Handle                */
```

```
                uchFileData,                /* String to be read           */
                100L,                       /* Length of string to be read */
                &ulBytesRead,               /* Bytes actually read         */
                FileHandleLock);            /* File handle lock id         */
   if (rc != NO_ERROR) {
      printf("DosProtectRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf("DosProtectRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
   if (rc != NO_ERROR) {
      printf("DosProtectClose error: return code = %u\n", rc);
      return 1;
    }
   return NO_ERROR;
}
```

DosProtectRead - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectSetFHState

DosProtectSetFHState - Syntax

Sets the state of the specified protected file handle.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;              /*  File handle to be set. */
ULONG     mode;               /*  Contents of the fsOpenMode field defined in a previous DosOpen. */
FHLOCK    fhFileHandleLockID; /*  The lockid of the protected file handle. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectSetFHState(hFile, mode, fhFileHandleLockID);
```

DosProtectSetFHState Parameter - hFile

hFile (HFILE) - input
File handle to be set.

DosProtectSetFHState Parameter - mode

mode (ULONG) - input
Contents of the *fsOpenMode* field defined in a previous DosOpen.

Possible modes are shown in the list below:

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) This bit must be set to 0.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Writes to the file may go through the system-buffer cache. 1 — Writes to the file may go through the system-buffer cache, but the data is written (the actual file I/O operation is completed) before a synchronous-write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous-write operations. This flag bit is not inherited by child processes.
13	OPEN_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands are always reported directly to the caller by way of a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. This flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00002000) Cache or No-Cache flag. The file is opened as follows: 0 — The disk driver should place data from I/O operations into cache. 1 — I/O operations to the file need not be done through the disk-driver cache. This bit is an advisory bit, and is used to advise file-system drivers and device drivers about whether the data should be cached. This bit, like the write-through bit, is a per-handle bit. This bit is not inherited by child processes.
11-8	These bits are reserved, and should be set to the values returned by DosQueryFHState in these positions.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 — File handle is inherited by a process created by DosExecPgm. 1 — File handle is private to the current process.
6-4	These bits must be set to 0. Any other values are invalid.
3	This bit is reserved, and should be set to the value returned by DosQueryFHState for this position.
2-0	These bits must be set to 0. Any other values are invalid.

DosProtectSetFHState Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The lockid of the protected file handle.

DosProtectSetFHState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosProtectSetFHState - Parameters

hFile (HFILE) - input
File handle to be set.

mode (ULONG) - input
Contents of the *fsOpenMode* field defined in a previous DosOpen.

Possible modes are shown in the list below:

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) This bit must be set to 0.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Writes to the file may go through the system-buffer cache. 1 — Writes to the file may go through the system-buffer cache, but the data is written (the actual file I/O operation is completed) before a synchronous-write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous-write operations. This flag bit is not inherited by child processes.
13	OPEN_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands are always reported directly to the caller by way of a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. This flag bit is not inherited by child processes.

12	OPEN_FLAGS_NO_CACHE (0x00002000) Cache or No-Cache flag. The file is opened as follows: 0 The disk driver should place data from I/O operations into cache. 1 I/O operations to the file need not be done through the disk-driver cache. This bit is an advisory bit, and is used to advise file-system drivers and device drivers about whether the data should be cached. This bit, like the write-through bit, is a per-handle bit. This bit is not inherited by child processes.
11-8	These bits are reserved, and should be set to the values returned by DosQueryFHState in these positions.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 File handle is inherited by a process created by DosExecPgm. 1 File handle is private to the current process.
6-4	These bits must be set to 0. Any other values are invalid.
3	This bit is reserved, and should be set to the value returned by DosQueryFHState for this position.
2-0	These bits must be set to 0. Any other values are invalid.

fhFileHandleLockID (FHLOCK) - input
The lockid of the protected file handle.

ulrc (APIRET) - returns
Return Code.

DosProtectFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosProtectSetFHState - Remarks

The operating system does not guarantee the write order for multiple-sector write operations. If an application requires several sectors to be written in a specific order, the operator should issue the sectors as separate synchronous-write operations. Setting the Write-Through flag does not affect any previous write operation. That data can remain in the buffers.

When the application cannot handle a critical error that occurs, critical-error handling can be reset to the system. This is done by having DosProtectFHState turn off the fail/errors bit, and then reissuing the I/O operation. The expected critical error recurs, and control is passed to the system critical-error handler. The precise time that the effect of this function is visible at the application level is unpredictable when asynchronous I/O operations are pending.

The file-handle-state bits set by this function can be queried by DosQueryFHState.

Named-Pipe Considerations

With DosProtectFHState, the inheritance (I) bit and Write-Through (W) bit can be set or reset. Setting W to 1 prevents write-behind operations on remote pipes.

DosProtectSetFHState - Related Functions

Related Functions

- DosDevIOCtl

- DosDupHandle
- DosExecPgm
- DosProtectClose
- DosProtectOpen
- DosProtectQueryFHState

DosProtectSetFHState - Example Code

This example queries and sets the file handle state of a protected file named "DOSPQFH.DAT".

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

UCHAR       uchFileName[]   = "DOSPQFH.DAT";   /* File to manipulate        */
HFILE       fhQryFile       = 0;               /* File handle from DosOpen  */
FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpen */
ULONG       FHState         = 0;                 /* File Handle State       */
APIRET      rc              = NO_ERROR;          /* Return code             */
FHLOCK      FileHandleLock  = 0;                 /* File handle lock        */

 rc = DosProtectOpen(uchFileName, &fhQryFile,
             &ulOpenAction, 10L, FILE_NORMAL,
             OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
             OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L,
             &FileHandleLock);
 if (rc != NO_ERROR) {
    printf("DosProtectOpen error: return code = %u\n", rc);
    return 1;
 }

 rc = DosProtectQueryFHState(fhQryFile, &FHState, FileHandleLock);
 if (rc != NO_ERROR) {
     printf("DosProtectQueryFHState error: return code = %u\n", rc);
     return 1;
 } else printf("FHState is: %x\n", FHState);

   /*   Change state to indicate that data should not be cached */

   FHState &= 0x7F88;                  /* Turn off non-participating bits */
   rc = DosProtectSetFHState(fhQryFile, FHState | OPEN_FLAGS_NO_CACHE,
                             FileHandleLock);
   if (rc != NO_ERROR) {
       printf("DosProtectSetFHState error: return code = %u\n", rc);
       return 1;
   }

   rc = DosProtectClose(fhQryFile, FileHandleLock);
   /* Should check if (rc != NO_ERROR) here */

   rc = DosDelete(uchFileName);      /* Delete the file */
   if (rc != NO_ERROR) {
       printf("DosDelete error: return code = %u\n", rc);
       return 1;
   } else {
       printf("File %s has been deleted.\n",uchFileName);
   } /* endif */

  return NO_ERROR;
}
```

DosProtectSetFHState - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectSetFileInfo

DosProtectSetFileInfo - Syntax

Sets file information.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hf;                 /*  File handle. */
ULONG     ulInfoLevel;        /*  Level of file information being set. */
PVOID     pInfoBuf;           /*  Address of the storage area containing the structures for file information le
ULONG     cbInfoBuf;          /*  The length, in bytes, of pInfoBuf. */
FHLOCK    fhFileHandleLockID; /*  The filehandle lockid obtained from DosProtectOpen. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectSetFileInfo(hf, ulInfoLevel,
         pInfoBuf, cbInfoBuf, fhFileHandleLockID);
```

DosProtectSetFileInfo Parameter - hf

hf (HFILE) - input
File handle.

DosProtectSetFileInfo Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
Level of file information being set.

A value of 1 or 2 can be specified, as shown in the following list.

1 FIL_STANDARD
Level 1 file information

2 FIL_QUERYEASIZE
Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

DosProtectSetFileInfo Parameter - pInfoBuf

pInfoBuf (PVOID) - input
Address of the storage area containing the structures for file information levels.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure.

Level 2 sets a series of EA name/value pairs. On input, *pInfoBuf* is an EAOP2 data structure. *fpGEA2List* is ignored. *fpFEA2List* points to a data area where the relevant is an FEA2 list is to be found. *oError* is ignored.

On output, *fpGEA2List* and *fpFEA2List* are unchanged. The area pointed to by *fpFEA2List* is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

DosProtectSetFileInfo Parameter - cbInfoBuf

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

DosProtectSetFileInfo Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

DosProtectSetFileInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectSetFileInfo returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFFER
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosProtectSetFileInfo - Parameters

hf (HFILE) - input
File handle.

ulInfoLevel (ULONG) - input
Level of file information being set.

A value of 1 or 2 can be specified, as shown in the following list.

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

pInfoBuf (PVOID) - input
Address of the storage area containing the structures for file information levels.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure.

Level 2 sets a series of EA name/value pairs. On input, *pInfoBuf* is an EAOP2 data structure. *fpGEA2List* is ignored. *fpFEA2List* points to a data area where the relevant is an FEA2 list is to be found. *oError* is ignored.

On output, *fpGEA2List* and *fpFEA2List* are unchanged. The area pointed to by *fpFEA2List* is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectSetFileInfo returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFFER
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosProtectSetFileInfo - Remarks

DosProtectSetFileInfo is successful only when the file is opened for write access, and access by other processes is prevented by a deny-both

sharing mode. If the file is already opened with conflicting sharing rights, any call to DosProtectOpen will fail.

A value of 0 in the date and time components of a field does not change the field. For example, if both "last write date" and "last write time" are specified as 0 in the Level 1 information structure, then both attributes of the file are left unchanged. If either "last write date" or "last write time" are other than 0, both attributes of the file are set to the new values.

In the FAT file system, only the dates and times of the last write can be modified. Creation and last-access dates and times are not affected.

The last-modification date and time will be changed if the extended attributes are modified.

DosProtectSetFileInfo - Related Functions

Related Functions

- DosProtectClose
- DosProtectEnumAttribute
- DosProtectOpen
- DosProtectQueryFileInfo
- DosQueryPathInfo
- DosResetBuffer
- DosSetFileSize
- DosSetPathInfo

DosProtectSetFileInfo - Example Code

This example creates a read-only file named "DOSFDEL.DAT", then changes its attributes to normal file, and finally uses DosForceDelete to delete the file so that it cannot be restore using UNDELETE.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

UCHAR       uchFileName[]   = "DOSFDEL.DAT";   /* File we want to delete    */
HFILE       fhDelFile       = 0;               /* File handle from DosOpen  */
FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
ULONG       ulBufferSize    = sizeof(FILESTATUS3); /* File info buffer size */
ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpen */
APIRET      rc              = NO_ERROR;          /* Return code             */
FHLOCK      FHLock          = 0;                 /* File handle lock        */

               /* Create a read-only file */

 rc = DosProtectOpen(uchFileName, &fhDelFile,
             &ulOpenAction, 10L, FILE_READONLY,
             OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
             OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L, &FHLock);
 if (rc != NO_ERROR) {
    printf("DosProtectOpen error: return code = %u\n", rc);
    return 1;
 }

 rc = DosProtectQueryFileInfo(fhDelFile, FIL_STANDARD,
             &fsts3FileInfo, ulBufferSize, FHLock);   /* Get standard info */
 if (rc != NO_ERROR) {
     printf("DosProtectQueryFileInfo error: return code = %u\n", rc);
     return 1;
 } else { printf("File %s created read-only.\n",uchFileName); }

   /*   Change the file attributes to be "normal"  */

   fsts3FileInfo.attrFile  = FILE_NORMAL;
   rc = DosProtectSetFileInfo(fhDelFile, FIL_STANDARD,
                       &fsts3FileInfo, ulBufferSize, FHLock);
   if (rc != NO_ERROR) {
       printf("DosProtectSetFileInfo error: return code = %u\n", rc);
```

```
        return 1;
    }

    rc = DosProtectClose(fhDelFile, FHLock);
    /* Should verify that (rc != NO_ERROR) here... */
           /* Delete the file */

    rc = DosForceDelete(uchFileName);
    if (rc != NO_ERROR) {
        printf("DosForceDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    }  /* endif */

   return NO_ERROR;
}
```

DosProtectSetFileInfo - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectSetFileLocks

DosProtectSetFileLocks - Syntax

Locks and unlocks a range of an open file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE        hFile;               /*  File handle. */
PFILELOCK    pflUnlock;           /*  Address of the structure containing the offset and length of a range to be
PFILELOCK    pflLock;             /*  Address of the structure containing the offset and length of a range to be
ULONG        timeout;             /*  The maximum time that the process is to wait for the requested locks. */
ULONG        flags;               /*  Flags that describe the action to be taken. */
FHLOCK       fhFileHandleLockID;  /*  The filehandle lockid returned by a previous DosProtectOpen. */
APIRET       ulrc;                /*  Return Code. */

ulrc = DosProtectSetFileLocks(hFile, pflUnlock,
         pflLock, timeout, flags, fhFileHandleLockID);
```

DosProtectSetFileLocks Parameter - hFile

hFile (HFILE) - input
File handle.

DosProtectSetFileLocks Parameter - pflUnlock

pflUnlock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be unlocked.

DosProtectSetFileLocks Parameter - pflLock

pflLock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be locked.

DosProtectSetFileLocks Parameter - timeout

timeout (ULONG) - input
The maximum time that the process is to wait for the requested locks.

The time is represented in milliseconds.

DosProtectSetFileLocks Parameter - flags

flags (ULONG) - input
Flags that describe the action to be taken.

Possible actions are shown in the list below:

Bit	Description
31-2	Reserved flags
1	Atomic This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, an error is returned. If this bit is set to 0, then the lock may or may not occur atomically with the unlock.
0	Share This bit defines the type of access that other processes may have to the file range that is being locked. If this bit is set to 0 (the default), other processes have no access to the locked file range. The

current process has exclusive access to the locked file range, which must not overlap any other locked file range.

If this bit is set to 1, the current process and other processes have shared read-only access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

DosProtectSetFileLocks Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned by a previous DosProtectOpen.

DosProtectSetFileLocks Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectSetFileLocks returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
174	ERROR_ATOMIC_LOCK_NOT_SUPPORTED
175	ERROR_READ_LOCKS_NOT_SUPPORTED

For a full list of error codes, see Errors.

DosProtectSetFileLocks - Parameters

hFile (HFILE) - input
File handle.

pflUnlock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be unlocked.

pflLock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be locked.

timeout (ULONG) - input
The maximum time that the process is to wait for the requested locks.

The time is represented in milliseconds.

flags (ULONG) - input
Flags that describe the action to be taken.

Possible actions are shown in the list below:

Bit	Description
31-2	Reserved flags

1	Atomic This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, an error is returned. If this bit is set to 0, then the lock may or may not occur atomically with the unlock.
0	Share This bit defines the type of access that other processes may have to the file range that is being locked. If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range. If this bit is set to 1, the current process and other processes have shared read-only access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned by a previous DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectSetFileLocks returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
174	ERROR_ATOMIC_LOCK_NOT_SUPPORTED
175	ERROR_READ_LOCKS_NOT_SUPPORTED

For a full list of error codes, see Errors.

DosProtectSetFileLocks - Remarks

DosProtectSetFileLocks allows a process to lock and unlock a range in a file. The time during which a file range is locked should be short.

If the lock and unlock ranges are both zero, ERROR_LOCK_VIOLATION is returned to the caller.

If you only want to lock a file range, set the unlock file offset and the unlock range length to zero.

If you only want to unlock a file range, set the lock file offset and the lock range length to zero.

When the Atomic bit of *flags* is set to 0, and DosProtectSetFileLocks specifies a lock operation and an unlock operation, the unlock operation occurs first, and then the lock operation is performed. If an error occurs during the unlock operation, an error code is returned and the lock operation is not performed. If an error occurs during the lock operation, an error code is returned and the unlock remains in effect if it was successful.

The lock operation is atomic when all of these conditions are met:

- The Atomic bit is set to 1 in *flags*
- The unlock range is the same as the lock range
- The process has shared access to the file range, and has requested exclusive access to it; or the process has exclusive access to the file range, and has requested shared access to it.

Some file system drivers (FSDs) may not support atomic lock operations. Versions of the operating system prior to OS/2 Version 2.00 do not support atomic lock operations. If the application receives the error code ERROR_ATOMIC_LOCK_NOT_SUPPORTED, the application should unlock the file range and then lock it using a non-atomic operation (with the atomic bit set to 0 in *flags*). The application should also refresh its internal buffers before making any changes to the file.

If you issue DosProtectClose to close a file with locks still in effect, the locks are released in no defined sequence.

If you end a process with a file open, and you have locks in effect in that file, the file is closed and the locks are released in no defined sequence.

The locked range can be anywhere in the logical file. Locking beyond the end of the file is not an error. A file range to be locked exclusively must first be cleared of any locked file sub-ranges or overlapping locked file ranges.

If you repeat DosProtectSetFileLocks for the same file handle and file range, then you duplicate access to the file range. Access to locked file ranges is not duplicated across DosExecPgm. The proper method of using locks is to attempt to lock the file range, and to examine the return value.

The following table shows the level of access granted when the accessed file range is locked with an exclusive lock or a shared lock. "Owner" refers to a process that owns the lock. "Non-owner" refers to a process that does not own the lock.

Action	Exclusive Lock	Shared Lock
Owner read	Success	Success
Non-owner read	Wait for unlock. Return error code after time-out.	Success
Owner write	Success	Wait for unlock. Return error code after time-out.
Non-owner write	Wait for unlock. Return error code after time-out.	Wait for unlock. Return error code after time-out.

If only locking is specified, DosProtectSetFileLocks locks the specified file range using *pflLock* If the lock operation cannot be accomplished, an error is returned, and the file range is not locked.

After the lock request is processed, a file range can be unlocked using the *pflUnlock* parameter of another DosProtectSetFileLocks request. If unlocking cannot be accomplished, an error is returned.

Instead of denying read/write access to an entire file by specifying access and sharing modes with DosProtectOpen requests, a process attempts to lock only the range needed for read/write access and examines the error code returned.

Once a specified file range is locked exclusively, read and write access by another process is denied until the file range is unlocked. If both unlocking and locking are specified by DosProtectSetFileLocks, the unlocking operation is performed first, then locking is done.

DosProtectSetFileLocks - Related Functions

Related Functions

- DosCancelLockRequest
- DosDupHandle
- DosExecPgm
- DosProtectOpen

DosProtectSetFileLocks - Example Code

This example opens or creates and opens a file named "FLOCK.DAT" in protected mode, and updates it using file locks.

```
#define INCL_DOSFILEMGR       /* File Manager values */
#define INCL_DOSERRORS        /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

HFILE     FileHandle   = NULLHANDLE;  /* File handle */
```

```
ULONG     Action       = 0,           /* Action taken by DosOpen */
          Wrote        = 0,           /* Number of bytes written by DosWrite */
          i            = 0;           /* Loop index */
CHAR      FileData[40] = "Forty bytes of demonstration text data\r\n";
APIRET    rc           = NO_ERROR;    /* Return code */
FHLOCK    FHLock       = 0;           /* File handle lock   */
FILELOCK  LockArea     = {0},         /* Area of file to lock */
          UnlockArea   = {0};         /* Area of file to unlock */

rc = DosProtectOpen("flock.dat",                  /* File to open */
             &FileHandle,                  /* File handle */
             &Action,                      /* Action taken */
             4000L,                        /* File primary allocation */
             FILE_ARCHIVED,                /* File attributes */
             FILE_OPEN | FILE_CREATE,      /* Open function type */
             OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
             0L,                           /* No extended attributes */
             &FHLock);                     /* File handle lock */
if (rc != NO_ERROR) {                      /* If open failed */
   printf("DosProtectOpen error: return code = %u\n", rc);
   return 1;
}

LockArea.lOffset = 0L;             /* Start locking at beginning of file */
LockArea.lRange =  40L;            /* Use a lock range of 40 bytes       */

         /* Write 8000 bytes to the file, 40 bytes at a time */

for (i=0; i<200; ++i) {

  rc = DosProtectSetFileLocks(FileHandle,        /* File handle   */
                       &UnlockArea,       /* Unlock previous record (if any) */
                       &LockArea,         /* Lock current record */
                       2000L,             /* Lock time-out value of 2 seconds */
                       0L,                /* Exclusive lock, not atomic */
                       FHLock);           /* File handle lock */
  if (rc != NO_ERROR) {
     printf("DosProtectSetFileLocks error: return code = %u\n", rc);
     return 1;
  }
  rc = DosProtectWrite(FileHandle, FileData, sizeof(FileData), &Wrote, FHLock);
  if (rc != NO_ERROR) {
     printf("DosProtectWrite error: return code = %u\n", rc);
     return 1;
  }

   UnlockArea = LockArea;      /* Will unlock this record on next iteration */
   LockArea.lOffset += 40L;    /* Prepare to lock next record               */

  } /* endfor - 8000 bytes written */

rc = DosProtectClose(FileHandle,FHLock);    /* Close file, release any locks */
/* Should check if (rc != NO_ERROR) here .... */

return NO_ERROR;
}
```

DosProtectSetFileLocks - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectSetFilePtr

DosProtectSetFilePtr - Syntax

Moves the read or write pointer according to the type of move specified.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;               /*  The handle returned by a previous DosOpen function. */
LONG      ib;                  /*  The signed distance (offset) to move, in bytes. */
ULONG     method;              /*  The method of moving. */
PULONG    ibActual;            /*  Address of the new pointer location. */
FHLOCK    fhFileHandleLockID;  /*  The filehandle lockid returned by a previous DosProtectOpen. */
APIRET    ulrc;                /*  Return Code. */

ulrc = DosProtectSetFilePtr(hFile, ib, method,
         ibActual, fhFileHandleLockID);
```

DosProtectSetFilePtr Parameter - hFile

hFile (HFILE) - input
The handle returned by a previous DosOpen function.

DosProtectSetFilePtr Parameter - ib

ib (LONG) - input
The signed distance (offset) to move, in bytes.

DosProtectSetFilePtr Parameter - method

method (ULONG) - input
The method of moving.

This field specifies the location in the file at which the read/write pointer starts before adding the *ib* offset. The values and their meanings are as shown in the following list:

0 FILE_BEGIN
Move the pointer from the beginning of the file.

1 FILE_CURRENT
Move the pointer from the current location of the read/write pointer.

2 FILE_END
Move the pointer from the end of the file. Use this method to determine a file's size.

DosProtectSetFilePtr Parameter - ibActual

ibActual (PULONG) - output
Address of the new pointer location.

DosProtectSetFilePtr Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned by a previous DosProtectOpen.

DosProtectSetFilePtr Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectSetFilePtr returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
132	ERROR_SEEK_ON_DEVICE
131	ERROR_NEGATIVE_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE

For a full list of error codes, see Errors.

DosProtectSetFilePtr - Parameters

hFile (HFILE) - input
The handle returned by a previous DosOpen function.

ib (LONG) - input
The signed distance (offset) to move, in bytes.

method (ULONG) - input
The method of moving.

This field specifies the location in the file at which the read/write pointer starts before adding the *ib* offset. The values and their meanings are as shown in the following list:

0 FILE_BEGIN
Move the pointer from the beginning of the file.

1 FILE_CURRENT
Move the pointer from the current location of the read/write pointer.

2 FILE_END
Move the pointer from the end of the file. Use this method to determine a file's size.

ibActual (PULONG) - output
Address of the new pointer location.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid returned by a previous DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectSetFilePtr returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
132	ERROR_SEEK_ON_DEVICE
131	ERROR_NEGATIVE_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE

For a full list of error codes, see Errors.

DosProtectSetFilePtr - Remarks

The read/write pointer in a file is a signed 32-bit number. A negative value for *ib* moves the pointer backward in the file; a positive value moves it forward. DosProtectSetFilePtr cannot be used to move to a negative position in the file.

DosProtectSetFilePtr cannot be used for a character device or pipe.

DosProtectSetFilePtr - Related Functions

Related Functions

- DosProtectOpen
- DosProtectRead
- DosProtectSetFileSize
- DosProtectWrite

DosProtectSetFilePtr - Example Code

This example opens or creates and opens a file named "DOSPROT.DAT", writes a string to it, returns the file pointer to the beginning of the file, reads it, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;
   ULONG  ulAction       = 0;
   ULONG  ulBytesRead    = 0;
   ULONG  ulWrote        = 0;
   ULONG  ulLocal        = 0;
   UCHAR  uchFileName[20]  = "dosprot.dat",
          uchFileData[100] = " ";
   FHLOCK FileHandleLock = 0;        /* File handle lock   */
   APIRET rc             = NO_ERROR; /* Return code */

   /* Open the file test.dat.  Make it read/write, open it */
```

```
   /* if it already exists and create it if it is new.     */
   rc = DosProtectOpen(uchFileName,             /* File path name          */
                &hfFileHandle,                  /* File handle             */
                &ulAction,                      /* Action taken            */
                100L,                           /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL,    /* File attribute          */
                OPEN_ACTION_CREATE_IF_NEW |
                OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type      */
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file   */
                0L,                             /* No extended attribute   */
                &FileHandleLock);               /* File handle lock id     */
   if (rc != NO_ERROR) {
      printf("DosProtectOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosProtectOpen: Action taken = %u\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n3...\n2...\n1\n");

   rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
                  (PVOID) uchFileData,       /* String to be written         */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote,                  /* Bytes actually written       */
                  FileHandleLock);           /* File handle lock id          */
   if (rc != NO_ERROR) {
      printf("DosProtectWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosProtectWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosProtectSetFilePtr (hfFileHandle,    /* File Handle          */
                      0L,                      /* Offset               */
                      FILE_BEGIN,              /* Move from BOF        */
                      &ulLocal,                /* New location address */
                      FileHandleLock);         /* File handle lock id  */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosProtectRead (hfFileHandle,         /* File Handle                */
                uchFileData,                  /* String to be read          */
                100L,                         /* Length of string to be read */
                &ulBytesRead,                 /* Bytes actually read        */
                FileHandleLock);              /* File handle lock id        */
   if (rc != NO_ERROR) {
      printf("DosProtectRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf("DosProtectRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
   if (rc != NO_ERROR) {
      printf("DosProtectClose error: return code = %u\n", rc);
      return 1;
    }
   return NO_ERROR;
}
```

DosProtectSetFilePtr - Topics

Select an item:

Syntax
Parameters
Returns
Remarks

Example Code
Related Functions
Glossary

DosProtectSetFileSize

DosProtectSetFileSize - Syntax

Changes the size of a file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;              /*  Handle of the file whose size to be changed. */
ULONG     cbSize;             /*  New size, in bytes, of the file. */
FHLOCK    fhFileHandleLockID; /*  The filehandle lockid obtained from DosProtectOpen. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectSetFileSize(hFile, cbSize,
         fhFileHandleLockID);
```

DosProtectSetFileSize Parameter - hFile

hFile (HFILE) - input
Handle of the file whose size to be changed.

DosProtectSetFileSize Parameter - cbSize

cbSize (ULONG) - input
New size, in bytes, of the file.

DosProtectSetFileSize Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

DosProtectSetFileSize Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectSetFileSize returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
112	ERROR_DISK_FULL

For a full list of error codes, see Errors.

DosProtectSetFileSize - Parameters

hFile (HFILE) - input
Handle of the file whose size to be changed.

cbSize (ULONG) - input
New size, in bytes, of the file.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectSetFileSize returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
112	ERROR_DISK_FULL

For a full list of error codes, see Errors.

DosProtectSetFileSize - Remarks

When DosProtectSetFileSize is issued, the file must be open in a mode that allows write access.

The size of the open file can be truncated or extended. If the file size is being extended, the file system tries to allocate additional bytes in a contiguous (or nearly contiguous) space on the medium. The values of the new bytes are undefined.

DosProtectSetFileSize - Related Functions

Related Functions

- DosProtectOpen
- DosProtectQueryFileInfo
- DosQueryPathInfo

DosProtectSetFileSize - Example Code

This example writes to a file named "DOSPMAN.DAT", resets the buffer, and changes the size of the file using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   FHLOCK FileHandleLock = 0;      /* File handle lock   */

   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   UCHAR  uchFileName[20]  = "dospman.dat",     /* Name of file */
          uchFileData[4]   = "DATA";            /* Data to write to file */
   APIRET rc             = NO_ERROR;           /* Return code */

   /* Open the file dosman.dat.  Use an existing file or create a new */
   /* one if it doesn't exist.                                       */
   rc = DosProtectOpen(uchFileName, &hfFileHandle, &ulAction, 4L,
              FILE_ARCHIVED | FILE_NORMAL,
              OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
              OPEN_FLAGS_NOINHERIT | OPEN_SHARE_DENYNONE  |
              OPEN_ACCESS_READWRITE, 0L, &FileHandleLock);
   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   }

   rc = DosProtectWrite (hfFileHandle, (PVOID) uchFileData,
                  sizeof (uchFileData), &ulWrote, FileHandleLock);
   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   }

   rc = DosResetBuffer (hfFileHandle);
   if (rc != NO_ERROR) {
      printf("DosResetBuffer error: return code = %u\n", rc);
      return 1;
   } /* endif */

   rc = DosProtectSetFileSize (hfFileHandle, 8L, FileHandleLock);
   if (rc != NO_ERROR) {
      printf("DosSetFileSize error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosProtectSetFileSize - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosProtectWrite

DosProtectWrite - Syntax

Writes a specified number of bytes from a buffer to the specified file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;              /*  File handle from DosOpen. */
PVOID     pBuffer;            /*  Address of the buffer that contains the data to write. */
ULONG     cbWrite;            /*  Number of bytes to write. */
PULONG    pcbActual;          /*  Address of the variable to receive the number of bytes actually written. */
FHLOCK    fhFileHandleLockID; /*  The filehandle lockid obtained from DosProtectOpen. */
APIRET    ulrc;               /*  Return Code. */

ulrc = DosProtectWrite(hFile, pBuffer, cbWrite,
         pcbActual, fhFileHandleLockID);
```

DosProtectWrite Parameter - hFile

hFile (HFILE) - input
File handle from DosOpen.

DosProtectWrite Parameter - pBuffer

pBuffer (PVOID) - input
Address of the buffer that contains the data to write.

DosProtectWrite Parameter - cbWrite

cbWrite (ULONG) - input
Number of bytes to write.

DosProtectWrite Parameter - pcbActual

pcbActual (PULONG) - output
Address of the variable to receive the number of bytes actually written.

DosProtectWrite Parameter - fhFileHandleLockID

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

DosProtectWrite Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosProtectWrite returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK
29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE

For a full list of error codes, see Errors.

DosProtectWrite - Parameters

hFile (HFILE) - input
File handle from DosOpen.

pBuffer (PVOID) - input
Address of the buffer that contains the data to write.

cbWrite (ULONG) - input
Number of bytes to write.

pcbActual (PULONG) - output
Address of the variable to receive the number of bytes actually written.

fhFileHandleLockID (FHLOCK) - input
The filehandle lockid obtained from DosProtectOpen.

ulrc (APIRET) - returns
Return Code.

DosProtectWrite returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK

29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE

For a full list of error codes, see Errors.

DosProtectWrite - Remarks

DosProtectWrite begins to write at the current file-pointer position. The file pointer is automatically moved by read and write operations. It can be moved to a desired position by issuing DosProtectSetFilePtr.

If the specified file has been opened using the write-through flag, DosProtectWrite writes the data to the disk before returning. Upon return to the caller, *pcbActual* contains the number of bytes actually written.

If there is not enough space on the disk or diskette to write all of the bytes specified by *cbWrite* then DosProtectWrite does not write any bytes. Upon return to the caller, *pcbActual* contains zero.

A value of zero for *cbWrite* is not considered an error. No data transfer occurs, and there is no effect on the file or the file pointer.

If the file is read-only, the write operation to the file is not performed.

If you issue DosProtectOpen with the Direct Open flag set to 1 in the *fsOpenMode* parameter, you have direct access to an entire disk or diskette volume, independent of the file system. You must lock the logical volume before accessing it, and you must unlock the logical volume when you are finished accessing it. Issue DosDevIOCtl for Category 8, DSK_LOCKDRIVE to lock the logical volume, and for Category 8, DSK_UNLOCKDRIVE to unlock the logical volume. While the logical volume is locked, no other process can access it.

Named-Pipe Considerations

DosProtectWrite also is used to write bytes or messages to a named pipe.

Each write operation to a message pipe writes a message whose size is the length of the write operation. DosProtectWrite automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Write operations in blocking mode always write all requested bytes before returning.

In nonblocking mode, DosProtectWrite returns either with all bytes written or none written. DosProtectWrite returns with no bytes written when it would have to divide the message into blocks in order to complete the request. This can occur when there is not enough space left in the pipe, or when the pipe is currently being written to by another client. If this occurs, DosProtectWrite returns immediately with a value of zero for *pcbActual* indicating that no bytes were written.

For a byte pipe, if the number of bytes to be written exceeds the space available in the pipe, DosProtectWrite writes as many bytes as it can, and returns with the number of bytes actually written in *pcbActual*.

An attempt to write to a pipe whose other end has been closed returns ERROR_BROKEN_PIPE.

DosProtectWrite - Related Functions

Related Functions

- DosProtectOpen
- DosProtectRead
- DosProtectSetFilePtr

DosProtectWrite - Example Code

This example opens or creates and opens a file named "DOSPROT.DAT", writes to it, reads from it, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
```

```
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;
   ULONG  ulAction       = 0;
   ULONG  ulBytesRead    = 0;
   ULONG  ulWrote        = 0;
   ULONG  ulLocal        = 0;
   UCHAR  uchFileName[20]  = "dosprot.dat",
          uchFileData[100] = " ";
   FHLOCK FileHandleLock = 0;        /* File handle lock   */
   APIRET rc             = NO_ERROR; /* Return code */

   /* Open the file test.dat.  Make it read/write, open it */
   /* if it already exists and create it if it is new.     */
   rc = DosProtectOpen(uchFileName,             /* File path name          */
               &hfFileHandle,                   /* File handle             */
               &ulAction,                       /* Action taken            */
               100L,                            /* File primary allocation */
               FILE_ARCHIVED | FILE_NORMAL,     /* File attribute          */
               OPEN_ACTION_CREATE_IF_NEW |
               OPEN_ACTION_OPEN_IF_EXISTS,      /* Open function type      */
               OPEN_FLAGS_NOINHERIT |
               OPEN_SHARE_DENYNONE  |
               OPEN_ACCESS_READWRITE,           /* Open mode of the file   */
               0L,                              /* No extended attribute   */
               &FileHandleLock);                /* File handle lock id     */
   if (rc != NO_ERROR) {
      printf("DosProtectOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosProtectOpen: Action taken = %u\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n3...\n2...\n1\n");

   rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
                  (PVOID) uchFileData,       /* String to be written         */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote,                  /* Bytes actually written       */
                  FileHandleLock);           /* File handle lock id          */
   if (rc != NO_ERROR) {
      printf("DosProtectWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosProtectWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosProtectSetFilePtr (hfFileHandle,    /* File Handle          */
                      0L,                      /* Offset               */
                      FILE_BEGIN,              /* Move from BOF        */
                      &ulLocal,                /* New location address */
                      FileHandleLock);         /* File handle lock id  */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosProtectRead (hfFileHandle,         /* File Handle                */
                uchFileData,                  /* String to be read          */
                100L,                         /* Length of string to be read */
                &ulBytesRead,                 /* Bytes actually read        */
                FileHandleLock);              /* File handle lock id        */
   if (rc != NO_ERROR) {
      printf("DosProtectRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf("DosProtectRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
   if (rc != NO_ERROR) {
      printf("DosProtectClose error: return code = %u\n", rc);
      return 1;
    }
   return NO_ERROR;
}
```

DosProtectWrite - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPurgeQueue

DosPurgeQueue - Syntax

Purges a queue of all its elements.

```
#define INCL_DOSQUEUES
#include <os2.h>

HQUEUE    hq;    /*  The handle of the queue to be purged. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosPurgeQueue(hq);
```

DosPurgeQueue Parameter - hq

hq (HQUEUE) - input
The handle of the queue to be purged.

DosPurgeQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosPurgeQueue returns one of the following values:

0	NO_ERROR
330	ERROR_QUE_PROC_NOT_OWNED

337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosPurgeQueue - Parameters

hq (HQUEUE) - input
The handle of the queue to be purged.

ulrc (APIRET) - returns
Return Code.

DosPurgeQueue returns one of the following values:

0	NO_ERROR
330	ERROR_QUE_PROC_NOT_OWNED
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosPurgeQueue - Remarks

The server process issues DosPurgeQueue to empty a queue of all its elements. This function is not available to client processes.

Warning: This is an unconditional purge of all elements in the queue.

DosPurgeQueue - Related Functions

Related Functions

- DosCloseQueue
- DosCreateQueue
- DosOpenQueue
- DosPeekQueue
- DosQueryQueue
- DosReadQueue
- DosWriteQueue

DosPurgeQueue - Example Code

This example creates a queue, writes to it, queries it, purges it, and finally closes it.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSERRORS   /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

#define QUE_NAME "\\QUEUES\\PANDAWRITE\\LOCALQUEUE"
```

```
int main(VOID) {

   HQUEUE QueueHandle = NULLHANDLE;  /* Queue handle                    */
   CHAR   *DataBuffer = "";          /* Data to write to queue         */
   ULONG  ulNumElems  = 0L;          /* Number of elements on queue    */
   APIRET rc          = NO_ERROR;    /* Return code                    */

   rc = DosCreateQueue(&QueueHandle, /* Queue handle                    */
          QUE_LIFO |                 /* Last In, First Out ordering    */
          QUE_CONVERT_ADDRESS,       /* Do 16-bit to 32-bit conversion */
          QUE_NAME);                 /* Name of the queue              */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "Element 1 of 2";
   rc = DosWriteQueue (QueueHandle, 100L, strlen(DataBuffer),
                       (PVOID)DataBuffer, 0L);
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryQueue (QueueHandle, &ulNumElems);
   if (rc != NO_ERROR) {
      printf ("DosQueryQueue error: return code = %u\n", rc);
      return 1;
   } else { printf ("DosQueryQueue: %u elements\n", ulNumElems); }

   DataBuffer = "Element 2 of 2";
   rc = DosWriteQueue (QueueHandle, 200L, strlen(DataBuffer),
                       (PVOID)DataBuffer, 0L);
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryQueue (QueueHandle, &ulNumElems);
   if (rc != NO_ERROR) {
      printf ("DosQueryQueue error: return code = %u\n", rc);
      return 1;
   } else { printf ("DosQueryQueue: %u elements\n", ulNumElems); }

   rc = DosPurgeQueue (QueueHandle);
   if (rc != NO_ERROR) {
      printf ("DosPurgeQueue error: return code = %u\n", rc);
      return 1;        }
   rc = DosQueryQueue (QueueHandle, &ulNumElems);
   if (rc != NO_ERROR) {
      printf ("DosQueryQueue error: return code = %u\n", rc);
      return 1;
   } else { printf ("DosQueryQueue: %u elements\n", ulNumElems); }

   rc = DosCloseQueue(QueueHandle);      /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;       }

   return NO_ERROR;
}
```

DosPurgeQueue - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosPutMessage

DosPutMessage - Syntax

Sends a message to an output file or device.

```
#define INCL_DOSMISC
#include <os2.h>

HFILE     hfile;  /*  The handle of the output file or device. */
ULONG     cbMsg;  /*  The length, in bytes, of the message to be sent. */
PCHAR     pBuf;   /*  The buffer that contains the message to be sent. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosPutMessage(hfile, cbMsg, pBuf);
```

DosPutMessage Parameter - hfile

hfile (HFILE) - input
The handle of the output file or device.

DosPutMessage Parameter - cbMsg

cbMsg (ULONG) - input
The length, in bytes, of the message to be sent.

DosPutMessage Parameter - pBuf

pBuf (PCHAR) - input
The buffer that contains the message to be sent.

DosPutMessage Return Value - ulrc

ulrc (APIRET) - returns

Return Code.

DosPutMessage returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
87	ERROR_INVALID_PARAMETER
321	ERROR_MR_UN_PERFORM

For a full list of error codes, see Errors.

DosPutMessage - Parameters

hfile (HFILE) - input
The handle of the output file or device.

cbMsg (ULONG) - input
The length, in bytes, of the message to be sent.

pBuf (PCHAR) - input
The buffer that contains the message to be sent.

ulrc (APIRET) - returns
Return Code.

DosPutMessage returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
87	ERROR_INVALID_PARAMETER
321	ERROR_MR_UN_PERFORM

For a full list of error codes, see Errors.

DosPutMessage - Remarks

DosPutMessage sends a message that is currently in a buffer to an output file or device.

Screen width is assumed to be 80 characters. If a word would go past column 78, it is moved to the beginning (column 1) of a new line.

DosPutMessage assumes that the starting cursor position is column 1 when handling a word wrap. If the last character to be positioned on a line is a double-byte character, the character is not bisected.

DosPutMessage - Related Functions

Related Functions

- DosGetMessage
- DosInsertMessage
- DosQueryMessageCP

DosPutMessage - Example Code

This example sends a message to the output file "MYMSG.DAT", automatically wrapping if necessary.

```
#define INCL_DOSFILEMGR /* File Manager values */
#define INCL_DOSMISC    /* Miscellaneous values */
#define INCL_DOSERRORS  /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

HFILE   FileHandle       = NULLHANDLE;      /* File handle */
ULONG   ulAction         = 0;               /* Action taken by DosOpen */
UCHAR   uchDataArea[160] = "";              /* Message buffer */
APIRET  rc               = 0;               /* Return code */

rc = DosOpen ("MYMSG.DAT", &FileHandle, &ulAction, 120L, FILE_ARCHIVED,
              OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
              OPEN_FLAGS_NOINHERIT | OPEN_SHARE_DENYREADWRITE |
              OPEN_ACCESS_READWRITE, 0L);

strcpy (uchDataArea, "This is a sample message that is going to be written ");
strcat (uchDataArea, "to the message file.  It is longer than 80");
strcat (uchDataArea, " characters, so it should wrap.              ");

rc = DosPutMessage(FileHandle, strlen(uchDataArea), uchDataArea);
if (rc != NO_ERROR) {
   printf("DosPutMessage error: return code = %u\n", rc);
   return 1;
}

rc = DosClose(FileHandle);

return NO_ERROR;
}
```

DosPutMessage - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryAppType

DosQueryAppType - Syntax

Returns the application type of an executable file.

```
#define INCL_DOSMODULEMGR
#include <os2.h>
```

```
PSZ       pszName;  /*  An ASCIIZ string that contains the file name of the executable file for which the flags
PULONG    pFlags;
APIRET    ulrc;     /*  Return Code. */

ulrc = DosQueryAppType(pszName, pFlags);
```

DosQueryAppType Parameter - pszName

pszName (PSZ) - input
An ASCIIZ string that contains the file name of the executable file for which the flags are to be returned.

If the string appears to be a fully qualified path (that is, it contains a " : " in the second position, or it contains a " \ ", or both), then the file is located in the indicated drive:directory. If neither of these is true, and this file name is not found in the current directory, each drive:directory specification in the path defined in the current program's environment is searched for this file. Note that any extension (.xxx) is acceptable for the executable file name. If no extension is specified, a default extension of ".exe" is used.

DosQueryAppType Parameter - pFlags

pFlags (PULONG) - output
A doubleword that will contain flags denoting the application type, as determined by reading the executable file header specified by *pszName* . Note that the call sequence passes a pointer to a location in application memory to return the application type flags.

pFlags is defined as follows:

Bit	Description
2-0	Indicate the application type as specified in the header: 000 — FAPPTYP_NOTSPEC (0x00000000) Application type is not specified in the executable header. 001 — FAPPTYP_NOTWINDOWCOMPAT (0x00000001) Application type is not-window-compatible. 010 — FAPPTYP_WINDOWCOMPAT (0x00000002) Application type is window-compatible. 011 — FAPPTYP_WINDOWAPI (0x00000003) Application type is window-API.
3	FAPPTYP_BOUND (0x00000008) Set to 1 if the executable file has been "bound" (by the BIND command) as a Family API application. Bits 0, 1, and 2 still apply.
4	FAPPTYP_DLL (0x00000010) Set to 1 if the executable file is a dynamic link library (DLL) module. Bits 0, 1, 2, 3, and 5 will be set to 0.
5	FAPPTYP_DOS (0x00000020) Set to 1 if the executable file is in PC/DOS format. Bits 0, 1, 2, 3, and 4 will be set to 0.
6	FAPPTYP_PHYSDRV (0x00000040) Set to 1 if the executable file is a physical device driver.
7	FAPPTYP_VIRTDRV (0x00000080) Set to 1 if the executable file is a virtual device driver.
8	FAPPTYP_PROTDLL (0x00000100) Set to 1 if the executable file is a protected-memory dynamic link library module.

9-13	Reserved.
14	FAPPTYP_32BIT (0x00004000) Set to 1 for 32-bit executable files.
15	Reserved.

DosQueryAppType Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryAppType returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
11	ERROR_BAD_FORMAT
15	ERROR_INVALID_DRIVE
32	ERROR_SHARING_VIOLATION
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID

For a full list of error codes, see Errors.

DosQueryAppType - Parameters

pszName (PSZ) - input
An ASCIIZ string that contains the file name of the executable file for which the flags are to be returned.

If the string appears to be a fully qualified path (that is, it contains a " : " in the second position, or it contains a " \ ", or both), then the file is located in the indicated drive:directory. If neither of these is true, and this file name is not found in the current directory, each drive:directory specification in the path defined in the current program's environment is searched for this file. Note that any extension (.xxx) is acceptable for the executable file name. If no extension is specified, a default extension of ".exe" is used.

pFlags (PULONG) - output
A doubleword that will contain flags denoting the application type, as determined by reading the executable file header specified by *pszName* . Note that the call sequence passes a pointer to a location in application memory to return the application type flags.

pFlags is defined as follows:

Bit	Description
2-0	Indicate the application type as specified in the header:
	000 — FAPPTYP_NOTSPEC (0x00000000) Application type is not specified in the executable header.
	001 — FAPPTYP_NOTWINDOWCOMPAT (0x00000001) Application type is not-window-compatible.
	010 — FAPPTYP_WINDOWCOMPAT (0x00000002) Application type is window-compatible.
	011 — FAPPTYP_WINDOWAPI (0x00000003) Application type is window-API.
3	FAPPTYP_BOUND (0x00000008) Set to 1 if the executable file has been "bound" (by the BIND command) as a Family API application.

	Bits 0, 1, and 2 still apply.
4	FAPPTYP_DLL (0x00000010) Set to 1 if the executable file is a dynamic link library (DLL) module. Bits 0, 1, 2, 3, and 5 will be set to 0.
5	FAPPTYP_DOS (0x00000020) Set to 1 if the executable file is in PC/DOS format. Bits 0, 1, 2, 3, and 4 will be set to 0.
6	FAPPTYP_PHYSDRV (0x00000040) Set to 1 if the executable file is a physical device driver.
7	FAPPTYP_VIRTDRV (0x00000080) Set to 1 if the executable file is a virtual device driver.
8	FAPPTYP_PROTDLL (0x00000100) Set to 1 if the executable file is a protected-memory dynamic link library module.
9-13	Reserved.
14	FAPPTYP_32BIT (0x00004000) Set to 1 for 32-bit executable files.
15	Reserved.

ulrc (APIRET) - returns
Return Code.

DosQueryAppType returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
11	ERROR_BAD_FORMAT
15	ERROR_INVALID_DRIVE
32	ERROR_SHARING_VIOLATION
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID

For a full list of error codes, see Errors.

DosQueryAppType - Remarks

DosQueryAppType returns the application type of an executable file.

The Presentation Manager shell uses this function to determine the application type that is being executed.

The application type is specified at link time in the module definition file.

DosQueryAppType - Related Functions

Related Functions

- DosLoadModule
- DosQueryProcType

DosQueryAppType - Example Code

This example shows how to obtain the application type of an executable file.

```
#define INCL_DOSSESMGR   /* Session Manager values */
#define INCL_DOSERRORS   /* DOS error values       */
#include <os2.h>
#include <stdio.h>

int main (VOID) {
   PSZ      szAppName = "C:\\OS2\\SYSLOG.EXE";  /* Application name      */
   ULONG    AppType = 0;            /* Application type flags (returned) */
   APIRET   rc      = NO_ERROR;     /* Return code                       */

   rc = DosQueryAppType(szAppName, &AppType);
                    /* On successful return, the AppType      */
                    /*   variable contains a set of bit flags */
                    /*   that describe the application type   */
                    /*   of the specified executable file     */

   if (rc != NO_ERROR) {
      printf("DosQueryAppType error: return code = %u\n", rc);
      return 1;
   } else {
    printf("Appname = %s\n", szAppName);
    printf("Apptype = %d\n", AppType & FAPPTYP_EXETYPE);
    printf(" Window API?     %s\n", (AppType & FAPPTYP_WINDOWAPI) ? "Y" : "N");
    printf(" Window compat?  %s\n", (AppType & FAPPTYP_WINDOWCOMPAT) ? "Y" : "N");
    printf(" Family API?     %s\n", (AppType & FAPPTYP_BOUND) ? "Y" : "N");
    printf(" PC/DOS format?  %s\n", (AppType & FAPPTYP_DOS) ? "Y" : "N");
    printf(" DLL?            %s\n",
                     (AppType & (FAPPTYP_DLL | FAPPTYP_PROTDLL) ) ? "Y" : "N");
   }
   return NO_ERROR;
}
```

DosQueryAppType - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryCollate

DosQueryCollate - Syntax

Obtains a collating sequence table that resides in the country file.

```
#define INCL_DOSNLS
#include <os2.h>

ULONG           cb;    /*  The length, in bytes, of the data area (pch) provided by the caller. */
```

```
PCOUNTRYCODE    pcc;   /*  Pointer to the COUNTRYCODE structure in which the country code and code page are iden
PCHAR           pch;   /*  The data area where the collating sequence table is returned. */
PULONG          pcch;  /*  The length, in bytes, of the collating sequence table returned. */
APIRET          ulrc;  /*  Return Code. */

ulrc = DosQueryCollate(cb, pcc, pch, pcch);
```

DosQueryCollate Parameter - cb

cb (ULONG) - input
The length, in bytes, of the data area (*pch*) provided by the caller.

A length value of 256 bytes is sufficient.

DosQueryCollate Parameter - pcc

pcc (PCOUNTRYCODE) - input
Pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the collate table for the default system country code is returned.

If *codepage* is set to 0, the collate table for the current process code page of the caller is returned.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

DosQueryCollate Parameter - pch

pch (PCHAR) - output
The data area where the collating sequence table is returned.

The caller provides this data area. The input parameter *cb* specifies the length of this area.

If this area is too small to hold all of the available information, then as much information as possible is provided in the available space (in the order in which the data would appear). If the amount of data returned is not enough to fill the memory area provided by the caller, then the memory that is unaltered by the available data is zeroed out. The format of the information returned in this buffer is as follows:

1 Byte	Sort weight of ASCII (0)
1 Byte	Sort weight of ASCII (1)
...	(additional values in collating order)
1 Byte	Sort weight of ASCII (255)

DosQueryCollate Parameter - pcch

pcch (PULONG) - output
The length, in bytes, of the collating sequence table returned.

DosQueryCollate Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryCollate returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
399	ERROR_NLS_TABLE_TRUNCATED
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosQueryCollate - Parameters

cb (ULONG) - input
The length, in bytes, of the data area (*pch*) provided by the caller.

A length value of 256 bytes is sufficient.

pcc (PCOUNTRYCODE) - input
Pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the collate table for the default system country code is returned.

If *codepage* is set to 0, the collate table for the current process code page of the caller is returned.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

pch (PCHAR) - output
The data area where the collating sequence table is returned.

The caller provides this data area. The input parameter *cb* specifies the length of this area.

If this area is too small to hold all of the available information, then as much information as possible is provided in the available space (in the order in which the data would appear). If the amount of data returned is not enough to fill the memory area provided by the caller, then the memory that is unaltered by the available data is zeroed out. The format of the information returned in this buffer is as follows:

1 Byte	Sort weight of ASCII (0)
1 Byte	Sort weight of ASCII (1)
...	(additional values in collating order)
1 Byte	Sort weight of ASCII (255)

pcch (PULONG) - output
The length, in bytes, of the collating sequence table returned.

ulrc (APIRET) - returns
Return Code.

DosQueryCollate returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
399	ERROR_NLS_TABLE_TRUNCATED
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosQueryCollate - Remarks

DosQueryCollate obtains a collating sequence table (for characters 0x00 through 0xFF) that resides in the country file (the default name is COUNTRY.SYS). It is used by the SORT utility to sort text according to the collating sequence.

The collating table returned corresponds to the system country code or selected country code, and to the process code page or selected code page.

DosQueryCollate - Related Functions

Related Functions

- DosMapCase
- DosQueryCp
- DosQueryCtryInfo
- DosQueryDBCSEnv
- DosSetProcessCp

DosQueryCollate - Example Code

This example gets the collating sequence table for the current country and code page.

```
#define INCL_DOSNLS     /* National Language Support values */
#define INCL_DOSERRORS  /* DOS error values                 */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
COUNTRYCODE UserInfo       = {0};       /* Country and code page requested */
UCHAR       achColSeq[256] = {0};       /* Collating sequence              */
ULONG       ulSeqLen       = 0;         /* Length of sequence returned     */
ULONG       i              = 0,         /* Two loop indices                */
            j              = 0;
APIRET      rc             = NO_ERROR;  /* Return code                     */

   UserInfo.country = 0;      /* Request information about current country */
   UserInfo.codepage = 0;     /* ... and current code page                 */

   rc = DosQueryCollate(sizeof(achColSeq),   /* Length of output area      */
                        &UserInfo,           /* Country and codepage info  */
                        achColSeq,           /* Area for collating sequence*/
                        &ulSeqLen);          /* Length of data returned    */

   if (rc != NO_ERROR) {
       printf("DosQueryCollate error: return code = %u\n",rc);
       return 1;
   }
        /* Show the order of the first 128 characters in the sequence */

  if (ulSeqLen >= 8*16) {
    for (i = 0; i < 8; i++) {
      for (j = 0; j < 16; j++) {
```

```
         printf("%3u ", achColSeq[i*16+j]);
         } /* endfor (j) */
       printf("\n");
       } /* endfor (i) */
   } else {
      printf("Unable to show first 128 characters... only %u were returned.\n",
             ulSeqLen);
      return 1;
   }  /* endif */

   return NO_ERROR;
}
```

DosQueryCollate - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryCp

DosQueryCp - Syntax

Allows a process to query its current process code page and the prepared system code pages.

```
#define INCL_DOSNLS
#include <os2.h>

ULONG     cb;    /*  The length, in bytes, of arCP. */
PULONG    arCP;  /*  The returned data list. */
PULONG    pcCP;
APIRET    ulrc;  /*  Return Code. */

ulrc = DosQueryCp(cb, arCP, pcCP);
```

DosQueryCp Parameter - cb

cb (ULONG) - input
The length, in bytes, of *arCP*.

DosQueryCp Parameter - arCP

arCP (PULONG) - output
The returned data list.

The first ULONG is the current code page identifier of the calling process.

If one or two code pages have been prepared for the system, then the second ULONG is the first prepared code page, and the third ULONG is the second prepared code page.

If the data length is less than the number of bytes needed to return all of the prepared system code pages, then the returned list is truncated.

The code page identifiers have the values described in the following list:

437	United States
850	Multilingual
852	Latin 2 (Czechoslovakia, Hungary, Poland, Yugoslavia)
857	Turkish
860	Portuguese
861	Iceland
863	Canadian French
865	Nordic
932	Japan
934	Korea
936	People's Republic of China
938	Taiwan
942	Japan SAA
944	Korea SAA
946	People's Republic of China SAA
948	Taiwan SAA

Note: Code pages 932, 934, 936, 938, 942, 944, 946, and 948 are supported only with the Asian version of the operating system on Asian hardware.

DosQueryCp Parameter - pcCP

pcCP (PULONG) - output
The length, in bytes, of the returned data.

DosQueryCp Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryCp returns one of the following values:

0	NO_ERROR
473	ERROR_CPLIST_TOO_SMALL
474	ERROR_CP_NOT_MOVED

For a full list of error codes, see Errors.

DosQueryCp - Parameters

cb (ULONG) - input
The length, in bytes, of *arCP*.

arCP (PULONG) - output
The returned data list.

The first ULONG is the current code page identifier of the calling process.

If one or two code pages have been prepared for the system, then the second ULONG is the first prepared code page, and the third ULONG is the second prepared code page.

If the data length is less than the number of bytes needed to return all of the prepared system code pages, then the returned list is truncated.

The code page identifiers have the values described in the following list:

437	United States
850	Multilingual
852	Latin 2 (Czechoslovakia, Hungary, Poland, Yugoslavia)
857	Turkish
860	Portuguese
861	Iceland
863	Canadian French
865	Nordic
932	Japan
934	Korea
936	People's Republic of China
938	Taiwan
942	Japan SAA
944	Korea SAA
946	People's Republic of China SAA
948	Taiwan SAA

Note: Code pages 932, 934, 936, 938, 942, 944, 946, and 948 are supported only with the Asian version of the operating system on Asian hardware.

pcCP (PULONG) - output
The length, in bytes, of the returned data.

ulrc (APIRET) - returns
Return Code.

DosQueryCp returns one of the following values:

0	NO_ERROR
473	ERROR_CPLIST_TOO_SMALL
474	ERROR_CP_NOT_MOVED

For a full list of error codes, see Errors.

DosQueryCp - Remarks

DosQueryCp allows a process to query its current process code page and the prepared system code pages.

The process code page identifier previously set by DosSetProcessCp or inherited by the process is returned to the caller. An input list size of two bytes returns only the current process code page identifier. If no code pages have been prepared with the CODEPAGE command, a length of two and a current code page identifier value of zero are returned.

Note: If no CODEPAGE= statement exists in CONFIG.SYS, the operating system uses the first codepage that it finds in the COUNTRY.SYS file for that country.

The system code page identifiers are returned to the caller in the same order as they appear in the CODEPAGE command. The code page identifiers are returned in the following order:

1. The current code page of the process (one of the system code pages).
2. The primary (default) system code page.
3. The secondary system code page, if specified.

DosQueryCp - Related Functions

Related Functions

- DosMapCase
- DosQueryCollate
- DosQueryCtryInfo
- DosQueryDBCSEnv
- DosSetProcessCp

DosQueryCp - Example Code

This example gets the current code page and up to three other prepared code pages.

```
#define INCL_DOSNLS     /* National Language Support values */
#define INCL_DOSERRORS  /* DOS error values                 */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
   ULONG  aulCpList[8]  = {0},               /* Code page list        */
          ulBufSize     = 8 * sizeof(ULONG), /* Size of output list   */
          ulListSize    = 0,                 /* Size of list returned */
          indx          = 0;                 /* Loop index            */
   APIRET rc            = NO_ERROR;          /* Return code           */

   rc = DosQueryCp(ulBufSize,      /* Length of output code page list  */
                   aulCpList,      /* List of code pages               */
                   &ulListSize);   /* Length of list returned          */

   if (rc != NO_ERROR) {
      printf("DosQueryCp error: return code = %u\n",rc);
      return 1;
   } else {
      for (indx=0; indx < ulListSize/sizeof(ULONG); indx++)
         printf ("aulCpList[%u] = %u\n", indx, aulCpList[indx]);
   }

   return NO_ERROR;
}
```

DosQueryCp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryCtryInfo

DosQueryCtryInfo - Syntax

Obtains country-dependent formatting information that resides in the country file.

```
#define INCL_DOSNLS
#include <os2.h>

ULONG           cb;         /*  The length, in bytes, of the data area (pci) provided by the caller. */
PCOUNTRYCODE    pcc;        /*  Pointer to the COUNTRYCODE structure in which the country code and code page are
PCOUNTRYINFO    pci;        /*  A pointer to the COUNTRYINFO in which the country-dependent information is place
PULONG          pcbActual;  /*  A pointer to a ULONG in which country-dependent data is returned. */
APIRET          ulrc;       /*  Return Code. */

ulrc = DosQueryCtryInfo(cb, pcc, pci, pcbActual);
```

DosQueryCtryInfo Parameter - cb

cb (ULONG) - input
The length, in bytes, of the data area (*pci*) provided by the caller.

A length value of 40 bytes is sufficient.

DosQueryCtryInfo Parameter - pcc

pcc (PCOUNTRYCODE) - input
Pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the country information for the default system country code is used.

If *codepage* is set to 0, the country information for the current process code page of the caller is used.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

DosQueryCtryInfo Parameter - pci

pci (PCOUNTRYINFO) - output
A pointer to the COUNTRYINFO in which the country-dependent information is placed.

The caller provides this data area. The input parameter *cb* specifies the size of this area.

If this area is too small to hold all of the available information, then as much information as possible is provided in the available space (in the order in which the data would appear). If the amount of data returned is not enough to fill the memory area provided by the caller, then the memory that is unaltered by the available data is zeroed out. The format of the information returned in this buffer is as

follows:

codepage is reserved, and must be set to 0.

DosQueryCtryInfo Parameter - pcbActual

pcbActual (PULONG) - output
A pointer to a ULONG in which country-dependent data is returned.

DosQueryCtryInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryCtryInfo returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
399	ERROR_NLS_TABLE_TRUNCATED
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosQueryCtryInfo - Parameters

cb (ULONG) - input
The length, in bytes, of the data area (*pci*) provided by the caller.

A length value of 40 bytes is sufficient.

pcc (PCOUNTRYCODE) - input
Pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the country information for the default system country code is used.

If *codepage* is set to 0, the country information for the current process code page of the caller is used.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

pci (PCOUNTRYINFO) - output
A pointer to the COUNTRYINFO in which the country-dependent information is placed.

The caller provides this data area. The input parameter *cb* specifies the size of this area.

If this area is too small to hold all of the available information, then as much information as possible is provided in the available space (in the order in which the data would appear). If the amount of data returned is not enough to fill the memory area provided by the caller, then the memory that is unaltered by the available data is zeroed out. The format of the information returned in this buffer is as follows:

codepage is reserved, and must be set to 0.

pcbActual (PULONG) - output

A pointer to a ULONG in which country-dependent data is returned.

ulrc (APIRET) - returns
Return Code.

DosQueryCtryInfo returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
399	ERROR_NLS_TABLE_TRUNCATED
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosQueryCtryInfo - Remarks

DosQueryCtryInfo obtains country-dependent formatting information that resides in the country file (the default name is COUNTRY.SYS).

The country-dependent information returned corresponds to the system country code or selected country code, and to the process code page or selected code page.

DosQueryCtryInfo - Related Functions

Related Functions

- DosMapCase
- DosQueryCollate
- DosQueryCp
- DosQueryDBCSEnv
- DosSetProcessCp

DosQueryCtryInfo - Example Code

This example displays the current date in country-dependent format.

```
#define INCL_DOSNLS        /* National Language Support values */
#define INCL_DOSDATETIME   /* Date and time values */
#define INCL_DOSERRORS     /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

COUNTRYCODE  Country    = {0};   /* Country code info (0 = current country) */
COUNTRYINFO  CtryInfo   = {0};   /* Buffer for country-specific information */
ULONG        ulInfoLen  = 0;
DATETIME     DateTime   = {0};       /* Date and time information           */
APIRET       rc         = NO_ERROR;  /* Return code                         */

  rc = DosQueryCtryInfo(sizeof(CtryInfo), &Country,
                        &CtryInfo, &ulInfoLen);

  if (rc != NO_ERROR) {
      printf("DosQueryCtryInfo error: return code = %u\n",rc);
      return 1;
  }

  rc = DosGetDateTime(&DateTime);  /* Retrieve the current date and time  */
```

```
   if (rc != NO_ERROR) {
      printf ("DosGetDateTime error : return code = %u\n", rc);
      return 1;
   } else {
      switch (CtryInfo.fsDateFmt) {

      case(1):                                                  /* dd/mm/yy */
      printf("Today is %d%s%d%s%d\n", DateTime.day, CtryInfo.szDateSeparator,
              DateTime.month, CtryInfo.szDateSeparator, DateTime.year);
        break;

      case(2):                                                  /* yy/mm/dd */
      printf("Today is %d%s%d%s%d\n", DateTime.year, CtryInfo.szDateSeparator,
              DateTime.month, CtryInfo.szDateSeparator, DateTime.day);
        break;

      default:                                                  /* mm/dd/yy */
      printf("Today is %d%s%d%s%d\n", DateTime.month, CtryInfo.szDateSeparator,
              DateTime.day, CtryInfo.szDateSeparator, DateTime.year);
        break;

       } /* endswitch */
   }
     return NO_ERROR;
   }
```

DosQueryCtryInfo - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryCurrentDir

DosQueryCurrentDir - Syntax

Gets the full path of the current directory for the requesting process on the specified drive.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     disknum;  /*  The drive number. */
PBYTE     pBuf;     /*  Address of the full path of the current directory. */
PULONG    pcbBuf;   /*  Address of the length, in bytes, of the pBuf buffer. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosQueryCurrentDir(disknum, pBuf, pcbBuf);
```

DosQueryCurrentDir Parameter - disknum

disknum (ULONG) - input
The drive number.

The value 0 means the current drive, 1 means drive A, 2 means drive B, 3 means drive C, and so on. The maximum possible value is 26, which corresponds to drive Z.

DosQueryCurrentDir Parameter - pBuf

pBuf (PBYTE) - output
Address of the full path of the current directory.

DosQueryCurrentDir Parameter - pcbBuf

pcbBuf (PULONG) - in/out
Address of the length, in bytes, of the *pBuf* buffer.

Input	This field contains the length, in bytes, of the directory path buffer.
Output	If an error is returned because the buffer is too small, this field contains the required length, in bytes, of the buffer.

DosQueryCurrentDir Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryCurrentDir returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
26	ERROR_NOT_DOS_DISK
108	ERROR_DRIVE_LOCKED
111	ERROR_BUFFER_OVERFLOW

For a full list of error codes, see Errors.

DosQueryCurrentDir - Parameters

disknum (ULONG) - input

The drive number.

The value 0 means the current drive, 1 means drive A, 2 means drive B, 3 means drive C, and so on. The maximum possible value is 26, which corresponds to drive Z.

pBuf (PBYTE) - output
Address of the full path of the current directory.

pcbBuf (PULONG) - in/out
Address of the length, in bytes, of the *pBuf* buffer.

Input	This field contains the length, in bytes, of the directory path buffer.
Output	If an error is returned because the buffer is too small, this field contains the required length, in bytes, of the buffer.

ulrc (APIRET) - returns
Return Code.

DosQueryCurrentDir returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
26	ERROR_NOT_DOS_DISK
108	ERROR_DRIVE_LOCKED
111	ERROR_BUFFER_OVERFLOW

For a full list of error codes, see Errors.

DosQueryCurrentDir - Remarks

The drive letter is not part of the returned string. The string does not begin with a backslash, and it ends with a byte containing 0x00.

The system provides the length of the returned path-name string in *pcbBuf*, which does not include the ending null byte. If the *pBuf* buffer is not large enough to hold the current-directory path string, the system returns the required length, in bytes, for the *pBuf* buffer in *pcbBuf*.

For file-system drivers, the case of the current directory is set at the time of creation.

Programs running without the NEWFILES bit set are allowed to issue DosSetCurrentDir for a directory that is not in the 8.3 file-name format.

An application must issue DosQuerySysInfo to determine the maximum path length supported by the operating system. The returned value should be used to dynamically allocate buffers that are to be used to store paths.

DosQueryCurrentDir - Related Functions

Related Functions

- DosQueryCurrentDisk
- DosQuerySysInfo
- DosSetCurrentDir
- DosSetDefaultDisk

DosQueryCurrentDir - Example Code

This example shows how to create, use, and then delete a new directory named "TEMPPROG". Some return code checking has been omitted for brevity.

```
#define INCL_DOSFILEMGR   /* File Manager values */
```

```
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   UCHAR   achOrigDirName[256] = "";            /* Original directory name  */
   UCHAR   achNewDirName[256]  = "\\TEMPPROG";  /* New directory to create  */
   UCHAR   achDirName[256]     = "";          /* Directory name for queries */
   ULONG   cbDirPathLen    = 0;               /* Length of directory path   */
   PEAOP2  pEABuf       = NULL;   /* Extended Attribute buffer pointer      */
   ULONG   ulDriveNum   = 0;      /* Drive number: current=0, A=1, B=2, ... */
   APIRET  rc           = NO_ERROR;      /* Return code                      */

   cbDirPathLen = (ULONG) sizeof(achOrigDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achOrigDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Original dir. = \\%s\n", achOrigDirName);

   pEABuf = NULL;  /* Indicate no EAs are to be defined for the directory  */
   rc = DosCreateDir(achNewDirName, pEABuf);   /* Create the new directory */

   if (rc != NO_ERROR) {
      printf("DosCreateDir error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir (achNewDirName);   /* Change to new directory     */

   ulDriveNum = 0;
   cbDirPathLen = (ULONG) sizeof(achDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Current dir.  = \\%s\n", achDirName);

   strcpy(achDirName,"\\");

   rc = DosSetCurrentDir (achDirName);     /* Change to root directory     */
   rc = DosDeleteDir (achNewDirName);      /* Delete the new directory     */

   return NO_ERROR;
}
```

DosQueryCurrentDir - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryCurrentDisk

DosQueryCurrentDisk - Syntax

Gets the current default drive for the requesting process.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PULONG    pdisknum;  /*  Address of the number of the default drive. */
PULONG    plogical;  /*  Address of the bit map where the system returns the mapping of the logical drives. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosQueryCurrentDisk(pdisknum, plogical);
```

DosQueryCurrentDisk Parameter - pdisknum

pdisknum (PULONG) - output
Address of the number of the default drive.

The value 1 means drive A, 2 means drive B, 3 means drive C, and so on. The maximum possible value is 26, which corresponds to drive Z.

DosQueryCurrentDisk Parameter - plogical

plogical (PULONG) - output
Address of the bit map where the system returns the mapping of the logical drives.

This bit map is stored in the low-order portion of the 32-bit area.

Logical drives A to Z have one-to-one mapping with bit positions 0 to 25 of the map; for example, bit 0 represents drive A, bit 1 represents drive B, and so on. The settings of these bits indicate which drives exist, as follows:

0	The logical drive does not exist.
1	The logical drive exists.

DosQueryCurrentDisk Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryCurrentDisk returns the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosQueryCurrentDisk - Parameters

pdisknum (PULONG) - output
Address of the number of the default drive.

The value 1 means drive A, 2 means drive B, 3 means drive C, and so on. The maximum possible value is 26, which corresponds to drive Z.

plogical (PULONG) - output
Address of the bit map where the system returns the mapping of the logical drives.

This bit map is stored in the low-order portion of the 32-bit area.

Logical drives A to Z have one-to-one mapping with bit positions 0 to 25 of the map; for example, bit 0 represents drive A, bit 1 represents drive B, and so on. The settings of these bits indicate which drives exist, as follows:

0	The logical drive does not exist.
1	The logical drive exists.

ulrc (APIRET) - returns
Return Code.

DosQueryCurrentDisk returns the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosQueryCurrentDisk - Related Functions

Related Functions

- DosSetDefaultDisk

DosQueryCurrentDisk - Example Code

This example displays information about the current disk and logical drives.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   ULONG   ulDriveNum   = 0;      /* Drive number (A=1, B=2, C=3, ...)    */
   ULONG   ulDriveMap   = 0;      /* Mapping of valid drives              */
   ULONG   i            = 0;      /* A loop index                         */
   APIRET  rc           = NO_ERROR;  /* Return code                       */

   rc = DosQueryCurrentDisk (&ulDriveNum, &ulDriveMap);

   if (rc != NO_ERROR) {
      printf ("DosQueryCurrentDisk error : return code = %u\n", rc);
      return 1;
   }
   printf ("Current disk = %c\n", 'A' + ulDriveNum - 1);
   printf ("Logical disks: ");
   printf ("A B C D E F G H I J K L M N O P Q R S T U V W X Y Z\n");
   printf ("        valid? ");

     /* Each bit in the ulDriveMap corresponds to a specific logical drive.
        bit 0 = A:, bit 1 = B:, ... , bit 24 = Y:, bit 25 = Z:
        For each drive, shift the bit string to the left to get rid of
        the bits before the one we want, then shift that result right 31
```

```
        bits to leave just the one we are interested in. */

  for (i = 0; i < 26; i++) {
     printf (( (ulDriveMap<<(31-i)) >> 31) ? "Y " : "- ");
    }
   printf ("\n");                          /* Print a newline character */

   return NO_ERROR;
}
```

DosQueryCurrentDisk - Topics

Select an item:

Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosQueryDBCSEnv

DosQueryDBCSEnv - Syntax

Obtains a DBCS (double-byte character set) environmental vector that resides in the country file.

```
#define INCL_DOSNLS
#include <os2.h>

ULONG           cb;     /*  The length, in bytes, of the data area (pBuf) provided by the caller. */
PCOUNTRYCODE    pcc;    /*  A pointer to the COUNTRYCODE structure in which the country code and code page are id
PCHAR           pBuf;   /*  The data area where the country-dependent information for the DBCS environmental vect
APIRET          ulrc;   /*  Return Code. */

ulrc = DosQueryDBCSEnv(cb, pcc, pBuf);
```

DosQueryDBCSEnv Parameter - cb

cb (ULONG) - input
The length, in bytes, of the data area (*pBuf*) provided by the caller.

A length value of 12 bytes is sufficient. The caller can always determine if all of the information has been obtained, because it ends with four bytes of 0s. A length of 4 is sufficient for information returned from non-DBCS-related countries.

DosQueryDBCSEnv Parameter - pcc

pcc (PCOUNTRYCODE) - input
A pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the DBCS information for the default system country code is returned.

If *codepage* is set to 0, the DBCS information for the current process code page of the caller is returned.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

DosQueryDBCSEnv Parameter - pBuf

pBuf (PCHAR) - output
The data area where the country-dependent information for the DBCS environmental vector is returned.

The caller provides this memory area. The size of the area is specified by the input parameter *cb* .

If this area is too small to hold all of the available information, then as much information as possible is provided in the available space (in the order in which the data would appear). Assuming that the data area is large enough, the valid information is terminated by two bytes of 0. The format of the information returned in this buffer is as follows:

2 Bytes	First range definition for DBCS lead byte values:	
	Byte 1	binary start value (inclusive) for range one
	Byte 2	binary stop value (inclusive) for range one
2 Bytes	Second range definition:	
	Byte 1	binary start value for range two
	Byte 2	binary stop value for range two
2 Bytes	Nth range definition:	
	Byte 1	binary start value for Nth range
	Byte 2	binary stop value for Nth range
2 Bytes	Two bytes of binary 0 end the list.	

```
For example: DB   81H,9FH
             DB   E0H,FCH
             DB   0,0
```

DosQueryDBCSEnv Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryDBCSEnv returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
399	ERROR_NLS_TABLE_TRUNCATED
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosQueryDBCSEnv - Parameters

cb (ULONG) - input
The length, in bytes, of the data area (*pBuf*) provided by the caller.

A length value of 12 bytes is sufficient. The caller can always determine if all of the information has been obtained, because it ends with four bytes of 0s. A length of 4 is sufficient for information returned from non-DBCS-related countries.

pcc (PCOUNTRYCODE) - input
A pointer to the COUNTRYCODE structure in which the country code and code page are identified.

If *country* is set to 0, the DBCS information for the default system country code is returned.

If *codepage* is set to 0, the DBCS information for the current process code page of the caller is returned.

Refer to the COUNTRYCODE for a table of values for country code and code page identifier.

pBuf (PCHAR) - output
The data area where the country-dependent information for the DBCS environmental vector is returned.

The caller provides this memory area. The size of the area is specified by the input parameter *cb*.

If this area is too small to hold all of the available information, then as much information as possible is provided in the available space (in the order in which the data would appear). Assuming that the data area is large enough, the valid information is terminated by two bytes of 0. The format of the information returned in this buffer is as follows:

2 Bytes	First range definition for DBCS lead byte values:	
	Byte 1	binary start value (inclusive) for range one
	Byte 2	binary stop value (inclusive) for range one
2 Bytes	Second range definition:	
	Byte 1	binary start value for range two
	Byte 2	binary stop value for range two
2 Bytes	Nth range definition:	
	Byte 1	binary start value for Nth range
	Byte 2	binary stop value for Nth range
2 Bytes	Two bytes of binary 0 end the list.	

```
For example: DB   81H,9FH
             DB   E0H,FCH
             DB   0,0
```

ulrc (APIRET) - returns
Return Code.

DosQueryDBCSEnv returns one of the following values:

0	NO_ERROR
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
399	ERROR_NLS_TABLE_TRUNCATED
401	ERROR_NLS_TYPE_NOT_FOUND
476	ERROR_CODE_PAGE_NOT_FOUND

For a full list of error codes, see Errors.

DosQueryDBCSEnv - Remarks

DosQueryDBCSEnv obtains a double-byte character set environmental vector that resides in the country file (the default name is COUNTRY.SYS).

The vector returned corresponds to the system country code or selected country code, and to the process code page or selected code page.

A double-byte character set is for a code page that contains more than 256 characters. A DBCS data string may contain both SBCS (single-byte character set) and DBCS (double-byte character set) characters.

A DBCS character is two bytes in length. It contains a lead byte and a trail byte. A lead byte is in the range returned by DosQueryDBCSEnv. A trail byte is not restricted to any range. The trail byte always follows the lead byte in a DBCS character.

DosQueryDBCSEnv - Related Functions

Related Functions

- DosMapCase
- DosQueryCollate
- DosQueryCp
- DosQueryCtryInfo
- DosSetProcessCp

DosQueryDBCSEnv - Example Code

This example shows how to get the double-byte character set vector from the country file, and displays the first and second range definition values.

```
#define INCL_DOSNLS     /* National Language Support values */
#define INCL_DOSERRORS  /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
COUNTRYCODE  ctrycodeInfo      = {0};        /* Country code information */
UCHAR        uchDBCSInfo[12]   = {0};        /* DBCS information buffer */
APIRET       rc                = NO_ERROR;   /* A return code */

ctrycodeInfo.country = 0;    /* Current country */
ctrycodeInfo.codepage = 0;   /* Current codepage */

rc = DosQueryDBCSEnv(sizeof(uchDBCSInfo),   /* Size of buffer */
                     &ctrycodeInfo,         /* Country code information */
                     uchDBCSInfo);          /* DBCS information buffer */

if (rc != NO_ERROR) {
   printf("DosQueryDBCSEnv error: return code = %u\n", rc);
   return 1;
 } else {
      /* For non-DBCS countries, these will be 4 bytes of 0 */
   printf("DBCS 1st range definition: %2.2x %2.2x\n",
                           uchDBCSInfo[0], uchDBCSInfo[1]);
   printf("     2nd range definition: %2.2x %2.2x\n",
                           uchDBCSInfo[2], uchDBCSInfo[3]);
 } /* endif */


return NO_ERROR;
}
```

DosQueryDBCSEnv - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryEventSem

DosQueryEventSem - Syntax

Retrieves the post count for an event semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HEV       hev;        /*  The handle of the event semaphore to query. */
PULONG    pulPostCt;  /*  A pointer to the semaphore's post count. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosQueryEventSem(hev, pulPostCt);
```

DosQueryEventSem Parameter - hev

hev (HEV) - input
The handle of the event semaphore to query.

DosQueryEventSem Parameter - pulPostCt

pulPostCt (PULONG) - output
A pointer to the semaphore's post count.

The post count is the number of times DosPostEventSem has been called since the last time the event semaphore was in the reset state.

DosQueryEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosQueryEventSem - Parameters

hev (HEV) - input
The handle of the event semaphore to query.

pulPostCt (PULONG) - output
A pointer to the semaphore's post count.

The post count is the number of times DosPostEventSem has been called since the last time the event semaphore was in the reset state.

ulrc (APIRET) - returns
Return Code.

DosQueryEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosQueryEventSem - Remarks

DosQueryEventSem returns the post count for an event semaphore. The post count is the number of times that DosPostEventSem has been called since the last time the semaphore was in the reset state.

This function can be called by any thread in the process that created the semaphore. Other processes can also call this function, but they must first gain access to the semaphore by calling DosOpenEventSem.

DosQueryEventSem - Related Functions

Related Functions

- DosCloseEventSem
- DosCreateEventSem
- DosOpenEventSem
- DosPostEventSem
- DosResetEventSem
- DosWaitEventSem

DosQueryEventSem - Example Code

This example handles the client side of a pipe. It opens an existing named pipe and event semaphore, sends a message to the host, reads the host reply, queries the post count of the event semaphore and finally closes the event semaphore and pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   HEV      hev                   = NULLHANDLE; /* Event semaphore handle */
   PIPEINFO PipeBuffer[4]         = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   ULONG    ulPostCt              = 0;
   int      i                     = 0;
   PIPESEMSTATE infobuf[3]        = {{0}};

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to Pipe.\n");

   rc = DosOpenEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev);
   if (rc != NO_ERROR) {
      printf("DosOpenEventSem error: error code = %u\n", rc);
      return 1;
   }

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);  /* Make printf appear on display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }
   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PHOST: %s\n\n", message);

   rc = DosQueryEventSem(hev, &ulPostCt);
   if (rc != NO_ERROR) {
      printf("DosQueryEventSem error: return code = %u\n", rc);
      return 1;
    } else printf("Current post count value is %u\n", ulPostCt);

   rc = DosCloseEventSem(hev);
   if (rc != NO_ERROR) {
      printf("DosCloseEventSem error: error code = %u\n", rc);
      return 1;
   }
```

```
   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosQueryEventSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryExtLIBPATH

DosQueryExtLIBPATH - Syntax

Returns the current path to be searched before or after the system LIBPATH when locating DLLs.

```
#define INCL_DOSMISC
#include <os2.h>

PSZ       pszExtLIBPATH;  /*  Extended LIBPATH string. */
ULONG     flags;          /*  Flag indicating when the new path is searched. */
APIRET    ulrc;           /*  Return Code. */

ulrc = DosQueryExtLIBPATH(pszExtLIBPATH, flags);
```

DosQueryExtLIBPATH Parameter - pszExtLIBPATH

pszExtLIBPATH (PSZ) - output
Extended LIBPATH string.

If the buffer pointed to by this parameter is not large enough to hold the extended LIBPATH or an extended LIBPATH is not currently defined, this parameter returns a pointer to a NULL string.

DosQueryExtLIBPATH Parameter - flags

flags (ULONG) - output
Flag indicating when the new path is searched.

Possible values are described in the following list:

BEGIN_LIBPATH
The new path is searched before the LIBPATH.
END_LIBPATH
The new path is searched after the LIBPATH.

DosQueryExtLIBPATH Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryExtLIBPATH returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFER

For a full list of error codes, see Errors.

DosQueryExtLIBPATH - Parameters

pszExtLIBPATH (PSZ) - output
Extended LIBPATH string.

If the buffer pointed to by this parameter is not large enough to hold the extended LIBPATH or an extended LIBPATH is not currently defined, this parameter returns a pointer to a NULL string.

flags (ULONG) - output
Flag indicating when the new path is searched.

Possible values are described in the following list:

BEGIN_LIBPATH
The new path is searched before the LIBPATH.
END_LIBPATH
The new path is searched after the LIBPATH.

ulrc (APIRET) - returns
Return Code.

DosQueryExtLIBPATH returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFER

For a full list of error codes, see Errors.

DosQueryExtLIBPATH - Related Functions

Related Functions

- DosSetExtLIBPATH

DosQueryExtLIBPATH - Example Code

This example updates the paths to be searched before and after the system LIBPATH, when locating a DLL.

```
 #define INCL_DOSMISC
 #define INCL_DOSERRORS
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 UCHAR   uchBeginLIBPATH[512] = "";       /* Begin LIBPATH value returned */
 UCHAR   uchEndLIBPATH[512]   = "";       /* End LIBPATH value returned   */
 APIRET  rc                   = NO_ERROR;    /* Return code               */

    rc = DosSetExtLIBPATH("C:\\TOOL_X\\VERS_20\\DLL",
                          BEGIN_LIBPATH);    /* Add to beginning LIBPATH */
    if (rc != NO_ERROR) {
        printf("DosSetExtLIBPATH error: return code = %u\n", rc);
        return 1;
    }

    rc = DosSetExtLIBPATH("C:\\TOOL_X\\VERS_10\\DLL",
                          END_LIBPATH);      /* Add to ending LIBPATH   */
    if (rc != NO_ERROR) {
        printf("DosSetExtLIBPATH error: return code = %u\n", rc);
        return 1;
    }

    rc = DosQueryExtLIBPATH(uchBeginLIBPATH,
                            BEGIN_LIBPATH);   /* Query the BeginLIBPATH */
    if (rc != NO_ERROR) {
        printf("DosQueryExtLIBPATH error: return code = %u\n", rc);
        return 1;
    }

    rc = DosQueryExtLIBPATH(uchEndLIBPATH,
                            END_LIBPATH);     /* Query the EndLIBPATH   */
    if (rc != NO_ERROR) {
        printf("DosQueryExtLIBPATH error: return code = %u\n", rc);
        return 1;
    }

    printf(" BeginLIBPATH = %s\n", uchBeginLIBPATH);
    printf("   EndLIBPATH = %s\n", uchEndLIBPATH);

  return NO_ERROR;
}
```

DosQueryExtLIBPATH - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

DosQueryFHState

DosQueryFHState - Syntax

Queries the state of the specified file handle.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;  /*  Handle of the file to be queried. */
PULONG    pMode;  /*  Address of the contents of the open-mode word defined in a previous DosOpen function. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosQueryFHState(hFile, pMode);
```

DosQueryFHState Parameter - hFile

hFile (HFILE) - input
Handle of the file to be queried.

DosQueryFHState Parameter - pMode

pMode (PULONG) - output
Address of the contents of the open-mode word defined in a previous DosOpen function.

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag: 0 — *pszFileName* field in DosOpen represents a file to be opened normally. 1 — *pszFileName* is the "drive:" (such as C: or A:). It represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Write operations to the file go through the file system buffer cache. 1 — Write operations to the file may go through the file system buffer cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit is set to 1 because the data must be written to the medium for synchronous write operations. The Write-Through flag bit is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler.

1 Reported directly to the caller by a return code.

Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always are reported directly to the caller by a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions.

The Fail-Errors flag bit is not inherited by child processes.

12 OPEN_FLAGS_NO_CACHE (0x00001000)
Cache or No-Cache:

0 The disk driver should place data from I/O operations into the cache on this file.

1 I/O operations to the file need not be done through the disk-driver cache.

The setting of this bit determines whether it is worth caching the data for file-systems drivers and device drivers. This bit, like the Write-Through bit, is a per-handle bit.

This bit is not inherited by child processes.

11-8 Reserved bits.

7 OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag:

0 The file handle is inherited by a process that is created by issuing DosExecPgm.

1 The file handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing-Mode flags: Define the operations other processes can perform on the file:

001 OPEN_SHARE_DENYREADWRITE
Deny read and write access.

010 OPEN_SHARE_DENYWRITE
Deny write access.

011 OPEN_SHARE_DENYREAD
Deny read access.

100 OPEN_SHARE_DENYNONE
Deny neither read nor write access (deny none).

Any other value is invalid.

3 Reserved.

2-0 Access-Mode flags. File access is assigned as follows:

000 OPEN_ACCESS_READONLY
Read-only access.

001 OPEN_ACCESS_WRITEONLY
Write-only access.

010 OPEN_ACCESS_READWRITE
Read and write access.

Any other value is invalid.

DosQueryFHState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosQueryFHState - Parameters

hFile (HFILE) - input
Handle of the file to be queried.

pMode (PULONG) - output
Address of the contents of the open-mode word defined in a previous DosOpen function.

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag: 0 — *pszFileName* field in DosOpen represents a file to be opened normally. 1 — *pszFileName* is the "drive:" (such as C: or A:). It represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Write operations to the file go through the file system buffer cache. 1 — Write operations to the file may go through the file system buffer cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit is set to 1 because the data must be written to the medium for synchronous write operations. The Write-Through flag bit is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always are reported directly to the caller by a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. The Fail-Errors flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00001000) Cache or No-Cache: 0 — The disk driver should place data from I/O operations into the cache on this file. 1 — I/O operations to the file need not be done through the disk-driver cache. The setting of this bit determines whether it is worth caching the data for file-systems drivers and device drivers. This bit, like the Write-Through bit, is a per-handle bit. This bit is not inherited by child processes.
11-8	Reserved bits.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 — The file handle is inherited by a process that is created by issuing DosExecPgm. 1 — The file handle is private to the current process.

This bit is not inherited by child processes.

6-4 Sharing-Mode flags: Define the operations other processes can perform on the file:

001	OPEN_SHARE_DENYREADWRITE Deny read and write access.
010	OPEN_SHARE_DENYWRITE Deny write access.
011	OPEN_SHARE_DENYREAD Deny read access.
100	OPEN_SHARE_DENYNONE Deny neither read nor write access (deny none).

Any other value is invalid.

3 Reserved.

2-0 Access-Mode flags. File access is assigned as follows:

000	OPEN_ACCESS_READONLY Read-only access.
001	OPEN_ACCESS_WRITEONLY Write-only access.
010	OPEN_ACCESS_READWRITE Read and write access.

Any other value is invalid.

ulrc (APIRET) - returns
Return Code.

DosQueryFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosQueryFHState - Remarks

When the application cannot handle a critical error that occurs, critical-error handling can be reset to the system. This is done by having DosSetFHState turn off the fail/errors bit and then reissuing the I/O function. The expected critical error recurs, and control is passed to the system critical-error handler. The precise time that the effect of this function is visible at the application level is unpredictable when asynchronous I/O operations are pending.

The Direct Open bit parameter (OPEN_FLAGS_DASD) is the "Direct I/O flag." It provides an access mechanism to a disk or diskette volume independent of the file system. This mode should be used only by system programs and not by application programs.

Named-Pipe Considerations

As defined by the operating system, D = 0. Other bits are as defined by DosCreateNPipe (serving end), DosOpen (client end), or the last DosSetFHState.

DosQueryFHState - Related Functions

Related Functions

- DosDevIOCtl
- DosOpen

- DosSetFHState

DosQueryFHState - Example Code

This example queries and sets the handle state of the file "DOSQFH.DAT", and then displays it.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 UCHAR       uchFileName[]   = "DOSQFH.DAT";    /* File to manipulate        */
 HFILE       fhQryFile       = 0;               /* File handle from DosOpen  */
 FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
 ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpen */
 ULONG       FHState         = 0;                 /* File Handle State       */
 APIRET      rc              = NO_ERROR;          /* Return code             */

                 /* Create a file */

  rc = DosOpen(uchFileName, &fhQryFile,
               &ulOpenAction, 10L, FILE_NORMAL,
               OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
               OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L);
  if (rc != NO_ERROR) {
     printf("DosOpen error: return code = %u\n", rc);
     return 1;
  }

  rc = DosQueryFHState(fhQryFile, &FHState);
  if (rc != NO_ERROR) {
      printf("DosQueryFHState error: return code = %u\n", rc);
      return 1;
  } else printf("FHState is: %x\n", FHState);

    /*   Change state to indicate that data should not be cached */

    FHState &= 0x7F88;                  /* Turn off non-participating bits */
    rc = DosSetFHState(fhQryFile, FHState | OPEN_FLAGS_NO_CACHE);
    if (rc != NO_ERROR) {
        printf("DosSetFHState error: return code = %u\n", rc);
        return 1;
    }

    rc = DosClose(fhQryFile);
    /* Should check if (rc != NO_ERROR) here... */

    rc = DosDelete(uchFileName);      /* Delete the file */
    if (rc != NO_ERROR) {
        printf("DosDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    }

   return NO_ERROR;
}
```

DosQueryFHState - Topics

Select an item:
Syntax
Parameters
Returns
Remarks

Example Code
Related Functions
Glossary

DosQueryFileInfo

DosQueryFileInfo - Syntax

Gets file information.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hf;           /*  The file handle. */
ULONG     ulInfoLevel;  /*  Level of file information required. */
PVOID     pInfo;        /*  Address of the storage area where the system returns the requested level of file inf
ULONG     cbInfoBuf;    /*  The length, in bytes, of pInfo. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosQueryFileInfo(hf, ulInfoLevel, pInfo,
         cbInfoBuf);
```

DosQueryFileInfo Parameter - hf

hf (HFILE) - input
The file handle.

DosQueryFileInfo Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
Level of file information required.

A value of 1, 2, or 3 can be specified, as follows:

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information
3	FIL_QUERYEASFROMLIST Level 3 file information

Level 4 is reserved.

The structures described in *pInfo* indicate the information returned for each of these levels.

DosQueryFileInfo Parameter - pInfo

pInfo (PVOID) - output
Address of the storage area where the system returns the requested level of file information.

File information, where applicable, is at least as accurate as the most recent DosClose, DosResetBuffer, DosSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfo contains the FILESTATUS3 data structure, in which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfo contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG. On output, this field contains the size, in bytes, of the file's entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file's entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

Input	*pInfo* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 list defining the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be 0. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfo* is unchanged. The buffer pointed to by *fpFEA2List* is filled with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size of the entire EA set on disk for the file, even though only a subset of attributes was requested.

DosQueryFileInfo Parameter - cbInfoBuf

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfo* .

DosQueryFileInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryFileInfo returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL

130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosQueryFileInfo - Parameters

hf (HFILE) - input
The file handle.

ulInfoLevel (ULONG) - input
Level of file information required.

A value of 1, 2, or 3 can be specified, as follows:

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information
3	FIL_QUERYEASFROMLIST Level 3 file information

Level 4 is reserved.

The structures described in *pInfo* indicate the information returned for each of these levels.

pInfo (PVOID) - output
Address of the storage area where the system returns the requested level of file information.

File information, where applicable, is at least as accurate as the most recent DosClose, DosResetBuffer, DosSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfo contains the FILESTATUS3 data structure, in which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfo contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG. On output, this field contains the size, in bytes, of the file's entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file's entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

Input	*pInfo* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 list defining the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be 0. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfo* is unchanged. The buffer pointed to by *fpFEA2List* is filled with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size of the entire EA set on disk for the file, even though only a subset of attributes was requested.

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfo* .

ulrc (APIRET) - returns
Return Code.

DosQueryFileInfo returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosQueryFileInfo - Remarks

In the FAT file system, only date and time information contained in Level 1 file information can be modified. Zero is returned for the creation and access dates and times.

To return information contained in any of the file information levels, DosQueryFileInfo must be able to read the open file. DosQueryFileInfo works only when the file is opened for read access, with a deny-write sharing mode specified for access by other processes. If another process that has specified conflicting sharing and access modes has already opened the file, any call to DosOpen will fail.

DosEnumAttribute returns the lengths of EAs. This information can be used to calculate what size *pInfo* needs to be to hold full-extended-attribute (FEA) information returned by DosQueryFileInfo when Level 3 is specified. The size of the buffer is calculated as follows:

Four bytes (for *oNextEntryOffset*) +
One byte (for *fEA* +
One byte (for *cbName*) +
Two bytes (for *cbValue*) +
Value of *cbName* (for the name of the EA) +
One byte (for terminating NULL in *cbName*) +
Value of *cbValue* (for the value of the EA)

DosQueryFileInfo - Related Functions

Related Functions

- DosClose
- DosEnumAttribute
- DosOpen
- DosQueryPathInfo
- DosResetBuffer
- DosSetFileInfo
- DosSetFileSize
- DosSetPathInfo

DosQueryFileInfo - Example Code

This example obtains the information of the CONFIG.SYS file.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>
```

```
 int main(VOID) {
 UCHAR        uchFileName[80] = "C:\\CONFIG.SYS";  /* File to manipulate    */
 FILESTATUS3  fsts3ConfigInfo = {{0}};       /* Buffer for file information */
 ULONG        ulBufSize     = sizeof(FILESTATUS3);  /* Size of above buffer */
 HFILE        hfConfig      = 0;             /* Handle for Config file      */
 ULONG        ulOpenAction  = 0;             /* Action taken by DosOpen     */
 APIRET       rc            = NO_ERROR;      /* Return code                 */

  rc = DosOpen(uchFileName,                  /* File to open (path and name) */
               &hfConfig,                /* File handle returned         */
               &ulOpenAction,               /* Action taken by DosOpen      */
               0L,0L,        /* Primary allocation and attributes (ignored) */
               OPEN_ACTION_FAIL_IF_NEW |
               OPEN_ACTION_OPEN_IF_EXISTS,  /* Open an existing file only   */
               OPEN_FLAGS_NOINHERIT | OPEN_ACCESS_READONLY |
               OPEN_SHARE_DENYNONE,         /* Read access only             */
               0L);                         /* Extended attributes (ignored)*/

    if (rc != NO_ERROR) {
        printf("DosOpen error: return code = %u\n", rc);
        return 1;
    }

    rc = DosQueryFileInfo(hfConfig,   /* Handle of file                   */
                          FIL_STANDARD,  /* Request standard (Level 1) info */
                          &fsts3ConfigInfo, /* Buffer for file information  */
                          ulBufSize);    /* Size of buffer                  */
    if (rc != NO_ERROR) {
        printf("DosQueryFileInfo error: return code = %u\n", rc);
        return 1;
    }

    rc = DosClose(hfConfig);      /* Close the file  (check RC in real life) */
    printf("%s ---  File size: %u bytes\n",uchFileName, fsts3ConfigInfo.cbFile);
    printf("Last updated: %d/%d/%d; %d:%2.2d\n",
            fsts3ConfigInfo.fdateLastWrite.month,        /* Month           */
            fsts3ConfigInfo.fdateLastWrite.day,          /* Day             */
            (fsts3ConfigInfo.fdateLastWrite.year+1980L), /* Years since 1980 */
            fsts3ConfigInfo.ftimeLastWrite.hours,        /* Hours           */
            fsts3ConfigInfo.ftimeLastWrite.minutes);     /* Minutes         */

  return NO_ERROR;
}
```

DosQueryFileInfo - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryFSAttach

DosQueryFSAttach - Syntax

Obtains information about an attached file system (local or remote), or about a character device or pseudocharacter device attached to the file system.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ             pszDeviceName;    /*  A drive designation or the name of a character or pseudocharacter device. */
ULONG           ulOrdinal;        /*  An index into the list of character or pseudocharacter devices, or the set o
ULONG           ulFSAInfoLevel;   /*  Level of information returned in pfsqb. */
PFSQBUFFER2     pfsqb;            /*  Address of the buffer for returned information. */
PULONG          pcbBuffLength;    /*  Address of the length, in bytes, of the data buffer. */
APIRET          ulrc;             /*  Return Code. */

ulrc = DosQueryFSAttach(pszDeviceName, ulOrdinal,
         ulFSAInfoLevel, pfsqb, pcbBuffLength);
```

DosQueryFSAttach Parameter - pszDeviceName

pszDeviceName (PSZ) - input
A drive designation or the name of a character or pseudocharacter device.

If it is a drive designation, *pszDeviceName* is an ASCIIZ string consisting of a drive name followed by a colon. If it is a character or pseudocharacter device name, *pszDeviceName* is an ASCIIZ string consisting of a file name and the subdirectory, DEV. *pszDeviceName* is ignored if level 2 or 3 is specified for *ulFSAInfoLevel*.

DosQueryFSAttach Parameter - ulOrdinal

ulOrdinal (ULONG) - input
An index into the list of character or pseudocharacter devices, or the set of drives.

ulOrdinal always starts at 1. The ordinal position of an item in a list has no significance. *ulOrdinal* is used only to step through the list. The mapping from *ulOrdinal* to the item is volatile, and may change from one call to DosQueryFSAttach to the next. *ulOrdinal* is ignored if level 1 is specified for *ulFSAInfoLevel*.

DosQueryFSAttach Parameter - ulFSAInfoLevel

ulFSAInfoLevel (ULONG) - input
Level of information returned in *pfsqb*.

Possible levels are described in the following list:

1 FSAIL_QUERYNAME
Returns data for the drive or device name specified in *pszDeviceName*. *ulOrdinal* is ignored.

2 FSAIL_DEVNUMBER
Returns data for the entry in the list of character or pseudocharacter devices selected by *ulOrdinal*. *pszDeviceName* is ignored.

3 FSAIL_DRVNUMBER
Returns data for the entry in the list of drives selected by *ulOrdinal*. *pszDeviceName* is ignored.

DosQueryFSAttach Parameter - pfsqb

pfsqb (PFSQBUFFER2) - output
Address of the buffer for returned information.

Note: The *szName* is the file-system driver name exported by the file-system driver. This name is not necessarily the same as the file-system driver name in the boot sector.

For local character devices (*iType* = 1), *cbFSDName* = 0, and *szName* contains only a terminating null byte; *cbFSAData* = 0.

For local drives (*iType* = 3), *szName* contains the name of the file-system driver attached to the drive at the time of the call. This information changes dynamically. If the drive is attached to the kernel's resident file system, *szName* contains FAT or an unknown name. Since the resident file system gets attached to any disk that other file-system drivers refuse to mount, it is possible to have a disk that does not contain a recognizable file system, but yet gets attached to the resident file system. In this case, it is possible to detect the difference, and this information would help programs to preserve data on a disk that was not properly recognized.

DosQueryFSAttach Parameter - pcbBuffLength

pcbBuffLength (PULONG) - in/out
Address of the length, in bytes, of the data buffer.

Input	The address of the length, in bytes, of the return buffer (*pfsqb*).
Output	The length, in bytes, of the data returned in *pfsqb* by the file-system driver.

DosQueryFSAttach Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryFSAttach returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
259	ERROR_NO_MORE_ITEMS

For a full list of error codes, see Errors.

DosQueryFSAttach - Parameters

pszDeviceName (PSZ) - input
A drive designation or the name of a character or pseudocharacter device.

If it is a drive designation, *pszDeviceName* is an ASCIIZ string consisting of a drive name followed by a colon. If it is a character or pseudocharacter device name, *pszDeviceName* is an ASCIIZ string consisting of a file name and the subdirectory, DEV. *pszDeviceName* is ignored if level 2 or 3 is specified for *ulFSAInfoLevel*.

ulOrdinal (ULONG) - input
An index into the list of character or pseudocharacter devices, or the set of drives.

ulOrdinal always starts at 1. The ordinal position of an item in a list has no significance. *ulOrdinal* is used only to step through the list. The mapping from *ulOrdinal* to the item is volatile, and may change from one call to DosQueryFSAttach to the next. *ulOrdinal* is ignored if level 1 is specified for *ulFSAInfoLevel*.

ulFSAInfoLevel (ULONG) - input
Level of information returned in *pfsqb*.

Possible levels are described in the following list:

1	FSAIL_QUERYNAME Returns data for the drive or device name specified in *pszDeviceName*. *ulOrdinal* is ignored.
2	FSAIL_DEVNUMBER Returns data for the entry in the list of character or pseudocharacter devices selected by *ulOrdinal*. *pszDeviceName* is ignored.
3	FSAIL_DRVNUMBER Returns data for the entry in the list of drives selected by *ulOrdinal*. *pszDeviceName* is ignored.

pfsqb (PFSQBUFFER2) - output
Address of the buffer for returned information.

Note: The *szName* is the file-system driver name exported by the file-system driver. This name is not necessarily the same as the file-system driver name in the boot sector.

For local character devices (*iType* = 1), *cbFSDName* = 0, and *szName* contains only a terminating null byte; *cbFSAData* = 0.

For local drives (*iType* = 3), *szName* contains the name of the file-system driver attached to the drive at the time of the call. This information changes dynamically. If the drive is attached to the kernel's resident file system, *szName* contains FAT or an unknown name. Since the resident file system gets attached to any disk that other file-system drivers refuse to mount, it is possible to have a disk that does not contain a recognizable file system, but yet gets attached to the resident file system. In this case, it is possible to detect the difference, and this information would help programs to preserve data on a disk that was not properly recognized.

pcbBuffLength (PULONG) - in/out
Address of the length, in bytes, of the data buffer.

Input	The address of the length, in bytes, of the return buffer (*pfsqb*).
Output	The length, in bytes, of the data returned in *pfsqb* by the file-system driver.

ulrc (APIRET) - returns
Return Code.

DosQueryFSAttach returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
259	ERROR_NO_MORE_ITEMS

For a full list of error codes, see Errors.

DosQueryFSAttach - Remarks

DosQueryFSAttach returns information about all block devices, and all character and pseudocharacter devices. The subject of the information returned by this call changes frequently. Therefore, the information that this function returns may no longer be valid when you receive it.

The information returned for disks attached to the resident file system of the kernel can be used to determine:

- If the kernel recognized the disk as one attached to its file system, or
- If the kernel attached its file system to the disk because no other file-system drivers were attached to the disk.

This information can be important for a program that needs to know what file system is attached to the drive. A situation could arise where the file-system driver that recognizes a certain disk has not been loaded into the system. There is then a potential for the data on the disk to be destroyed because the wrong file system gets attached to the disk by default.

DosQueryFSAttach - Related Functions

Related Functions

- DosFSAttach
- DosQuerySysInfo

DosQueryFSAttach - Example Code

This example returns information about an attached file system.

```
 #define INCL_DOSFILEMGR   /* File manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>
 #include <string.h>

int main(VOID)
  {
  UCHAR  szDeviceName[8] = "C:";  /* Device name or drive letter string */
  ULONG  ulOrdinal       = 0;     /* Ordinal of entry in name list      */
  PBYTE  pszFSDName      = NULL;  /* pointer to FS name                 */
  PBYTE  prgFSAData      = NULL;  /* pointer to FS data                 */
  APIRET rc              = NO_ERROR; /* Return code                     */

    /* Return-data buffer should be large enough to hold FSQBUFFER2 */
    /* and the maximum data for szName, szFSDName, and rgFSAData    */
    /* Typically, the data isn't that large.                        */

  BYTE         fsqBuffer[sizeof(FSQBUFFER2) + (3 * CCHMAXPATH)] = {0};
  ULONG        cbBuffer   = sizeof(fsqBuffer);        /* Buffer length) */
  PFSQBUFFER2  pfsqBuffer = (PFSQBUFFER2) fsqBuffer;

    rc = DosQueryFSAttach(
                szDeviceName,    /* Logical drive of attached FS      */
                ulOrdinal,       /* ignored for FSAIL_QUERYNAME       */
                FSAIL_QUERYNAME, /* Return data for a Drive or Device */
                pfsqBuffer,      /* returned data                     */
                &cbBuffer);      /* returned data length              */

    /* On successful return, the fsqBuffer structure contains     */
    /* a set of information describing the specified attached     */
    /* file system and the DataBufferLen variable contains        */
    /* the size of information within the structure               */

    if (rc != NO_ERROR) {
       printf("DosQueryFSAttach error: return code = %u\n", rc);
       return 1;
    } else {

       /* The data for the last three fields in the FSQBUFFER2    */
       /* structure are stored at the offset of fsqBuffer.szName. */
       /* Each data field following fsqBuffer.szName begins       */
       /* immediately after the previous item.                    */

       pszFSDName = pfsqBuffer->szName + pfsqBuffer->cbName + 1;
       prgFSAData = pszFSDName + pfsqBuffer->cbFSDName + 1;

       printf("iType     = %d\n", pfsqBuffer->iType);
       printf("szName    = %s\n", pfsqBuffer->szName);
       printf("szFSDName = %s\n", pszFSDName);
```

```
      printf("rgFSAData = %s\n", prgFSAData);
    }
   return NO_ERROR;
}
```

DosQueryFSAttach - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryFSInfo

DosQueryFSInfo - Syntax

Gets information from a file-system device.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     disknum;    /*  Logical drive number for the disk about which information is to be retrieved. */
ULONG     infolevel;  /*  Level of file information required. */
PVOID     pBuf;       /*  Address of the storage area where the system returns the requested level of file infor
ULONG     cbBuf;      /*  The length, in bytes, of the buffer that receives the file-system information. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosQueryFSInfo(disknum, infolevel,
         pBuf, cbBuf);
```

DosQueryFSInfo Parameter - disknum

disknum (ULONG) - input
Logical drive number for the disk about which information is to be retrieved.

This parameter can be any value from 0 through 26. If this parameter is zero, information about the disk in the current drive is retrieved. Otherwise, 1 specifies drive A, 2 specifies drive B, and so on.

When a logical drive is specified, the media in the drive is examined (for a local drive only), and the request is passed to the file system driver (FSD) responsible for managing that media, or to the FSD that is attached to the drive.

DosQueryFSInfo Parameter - infolevel

infolevel (ULONG) - input
Level of file information required.

Possible levels are described in the list below:

1	FSIL_ALLOC Level 1 information. The information is returned in the format of an FSALLOCATE structure.
2	FSIL_VOLSER Level 2 information. The information is returned in the format of an FSINFO structure.

DosQueryFSInfo Parameter - pBuf

pBuf (PVOID) - output
Address of the storage area where the system returns the requested level of file information.

Level 1 Information
When a value of 1 (FSIL_ALLOC) is specified for *infolevel*, the information is returned in the format of an FSALLOCATE structure.

Level 2 Information
When a value of 2 (FSIL_VOLSER) is specified for *infolevel*, the information is returned in the format of an FSINFO structure.

DosQueryFSInfo Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of the buffer that receives the file-system information.

DosQueryFSInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryFSInfo returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
125	ERROR_NO_VOLUME_LABEL

For a full list of error codes, see Errors.

DosQueryFSInfo - Parameters

disknum (ULONG) - input
Logical drive number for the disk about which information is to be retrieved.

This parameter can be any value from 0 through 26. If this parameter is zero, information about the disk in the current drive is retrieved. Otherwise, 1 specifies drive A, 2 specifies drive B, and so on.

When a logical drive is specified, the media in the drive is examined (for a local drive only), and the request is passed to the file system driver (FSD) responsible for managing that media, or to the FSD that is attached to the drive.

infolevel (ULONG) - input
Level of file information required.

Possible levels are described in the list below:

1	FSIL_ALLOC Level 1 information
2	FSIL_VOLSER Level 2 information

pBuf (PVOID) - output
Address of the storage area where the system returns the requested level of file information.

Level 1 Information
When a value of 1 (FSIL_ALLOC) is specified for *infolevel*, the information is returned in the format of an FSALLOCATE structure.

Level 2 Information
When a value of 2 (FSIL_VOLSER) is specified for *infolevel*, the information is returned in the format of an FSINFO structure.

cbBuf (ULONG) - input
The length, in bytes, of the buffer that receives the file-system information.

ulrc (APIRET) - returns
Return Code.

DosQueryFSInfo returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
125	ERROR_NO_VOLUME_LABEL

For a full list of error codes, see Errors.

DosQueryFSInfo - Remarks

DosQueryFSInfo gets information from a file-system device.

If the disk or diskette has no volume label, the volume label is returned as a null string.

Trailing blanks supplied at the time the volume label is defined are not considered part of the label and are not returned by DosQueryFSInfo.

DosQueryFSInfo - Related Functions

Related Functions

- DosSetFSInfo

DosQueryFSInfo - Example Code

The first example displays the volume label associated with a particular drive. The second example provides allocation information for a drive from the file system.

```
 #define INCL_DOSFILEMGR   /* File manager values  */
 #define INCL_DOSERRORS    /* DOS error values     */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {

 ULONG       ulDriveNumber = 0;          /* Drive number */
 FSINFO      fsBuffer      = {0};        /* File system info buffer */
 APIRET      rc            = NO_ERROR;   /* Return code */

    ulDriveNumber = 3;          /* Specify drive C (A=1, B=2, C=3, ...) */

    rc = DosQueryFSInfo(ulDriveNumber,
                        FSIL_VOLSER,      /* Request volume information */
                        &fsBuffer,        /* Buffer for information     */
                        sizeof(FSINFO));     /* Size of buffer          */

    if (rc != NO_ERROR) {
        printf("DosQueryFSInfo error: return code = %u\n", rc);
        return 1;
    } else {
        printf("Volume label: '%s'\n", fsBuffer.vol.szVolLabel);
    }
    return NO_ERROR;
  }
```

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>

int main (VOID) {
   FSALLOCATE fsaBuffer     = {0};         /* File system info buffer     */
   APIRET  rc               = NO_ERROR;    /* Return code                 */

   rc = DosQueryFSInfo(3L,                     /* Drive number 3 (C:)     */
                       FSIL_ALLOC,             /* Level 1 allocation info */
                       (PVOID)&fsaBuffer,      /* Buffer                  */
                       sizeof(FSALLOCATE));    /* Size of buffer          */

   if (rc != NO_ERROR) {
      printf("DosQueryFSInfo error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("%121d bytes in each allocation unit.\n",
               fsaBuffer.cSectorUnit * fsaBuffer.cbSector);
             /* (Sectors per allocation unit) * (Bytes per sector) */
      printf ("%121d total allocation units.\n", fsaBuffer.cUnit);
      printf ("%121d available allocation units on disk.\n", fsaBuffer.cUnitAvail);
   }
   DosExit(EXIT_THREAD,fsaBuffer.cUnitAvail);  /* Return available allocation units
                                                  to the initiating process      */
   return NO_ERROR;
}
```

DosQueryFSInfo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryHType

DosQueryHType - Syntax

Determines whether a handle refers to a file or a device.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;  /*  File handle. */
PULONG    pType;  /*  Address of the value indicating the handle type. */
PULONG    pAttr;  /*  Address of the device-driver attribute word if pType indicates a local character device. *
APIRET    ulrc;   /*  Return Code. */

ulrc = DosQueryHType(hFile, pType, pAttr);
```

DosQueryHType Parameter - hFile

hFile (HFILE) - input
File handle.

DosQueryHType Parameter - pType

pType (PULONG) - output
Address of the value indicating the handle type.

Possible values are shown in the following list:

Bit	Description
15	Network bit:
	0 The handle refers to a local file, device, or pipe.
	1 The handle refers to a remote file, device, or pipe.
14	1 Protected file handle.

13-8	Reserved.
7-0	HandleClass: Describes the handle class. It may take on the following values in the low byte of *pType* :

0	Disk file
1	Character device
2	Pipe.

Values greater than 2 are reserved.

DosQueryHType Parameter - pAttr

pAttr (PULONG) - output
Address of the device-driver attribute word if *pType* indicates a local character device.

DosQueryHType Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryHType returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosQueryHType - Parameters

hFile (HFILE) - input
File handle.

pType (PULONG) - output
Address of the value indicating the handle type.

Possible values are shown in the following list:

Bit	Description
15	Network bit:

0	The handle refers to a local file, device, or pipe.
1	The handle refers to a remote file, device, or pipe.

14	1 Protected file handle.
13-8	Reserved.
7-0	HandleClass: Describes the handle class. It may take on the following values in the low byte of *pType* :

0	Disk file
1	Character device
2	Pipe.

Values greater than 2 are reserved.

pAttr (PULONG) - output
Address of the device-driver attribute word if *pType* indicates a local character device.

ulrc (APIRET) - returns
Return Code.

DosQueryHType returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosQueryHType - Remarks

DosQueryHType enables programs that are interactive or file-oriented to determine the source of their input. For example, CMD.EXE suppresses writing prompts if the input is from a disk file.

DosQueryHType - Related Functions

Related Functions

- DosOpen

DosQueryHType - Example Code

This example handles the client side of a pipe. It opens an existing named pipe, queries the pipe handle type and pipe state, sends a message to the host, reads the host reply, and finally closes the pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   PIPEINFO PipeBuffer[4]         = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   ULONG    PipeState             = 0;
   ULONG    HandType              = 0;
   ULONG    FlagWord              = 0;
   ULONG    BytesRead             = 0;

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
```

```
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to pipe.\n");

   rc = DosQueryHType(PipeHandle, &HandType, &FlagWord);
   if (rc != NO_ERROR) {
      printf("DosQueryHType error: error code = %u\n", rc);
      return 1;
   }  else printf("Handle type value is %u\n", HandType);
   rc = DosPeekNPipe(PipeHandle, Buffer, sizeof(Buffer),
                     &BytesRead, &BytesAvail, &PipeState);
   if (rc != NO_ERROR) {
      printf("DosPeekNPipe error: error code = %u\n", rc);
      return 1;
   }  else printf("Pipe status value is %u\n\n", PipeState);

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);   /* Flush above printf out to display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }

   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST: %s\n\n", message);

   rc = DosClose(PipeHandle);
   /* Should check if (rc != NO_ERROR) here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosQueryHType - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryMem

DosQueryMem - Syntax

Obtains information about a range of pages within the virtual-address space of the subject process.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pb;     /*  The base address of the range of pages to be queried. */
PULONG    pcb;    /*  A pointer to a ULONG containing the size of the range of pages. */
PULONG    pFlag;  /*  A pointer to a ULONG containing a set of attribute flags describing the type of allocation
APIRET    ulrc;   /*  Return Code. */

ulrc = DosQueryMem(pb, pcb, pFlag);
```

DosQueryMem Parameter - pb

pb (PVOID) - input
The base address of the range of pages to be queried.

DosQueryMem Parameter - pcb

pcb (PULONG) - in/out
A pointer to a ULONG containing the size of the range of pages.

Input	This parameter points to a variable that contains the size, in bytes, of the range of pages to be queried. The initial value of the variable is rounded to include all pages addressed by the requested base address and size.
Output	This parameter points to a variable that contains the actual size, in bytes, of the queried range of pages.

DosQueryMem Parameter - pFlag

pFlag (PULONG) - output
A pointer to a ULONG containing a set of attribute flags describing the type of allocation and access protection for the specified range of pages.

Allocation Type

PAG_COMMIT (0x00000010)
Pages within the specified region are committed.

PAG_SHARED (0x00002000)
Pages within the specified region are in a shared memory object. Otherwise, the pages are in a private memory object.

PAG_FREE (0x00004000)
Pages within the specified region are free.

PAG_BASE (0x00010000)
First page in the specified region is the first page in an allocated memory object.

Access Protection

PAG_READ (0x00000001)
Read access to the committed range of pages is allowed.

PAG_WRITE (0x00000002)
write access to the committed range of pages is allowed.

PAG_EXECUTE (0x00000004)
Execute access to the committed range of pages is allowed.

PAG_GUARD (0x00000008)
Access to the committed range of pages causes a "guard page entered" condition to be raised in the subject process.

DosQueryMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosQueryMem - Parameters

pb (PVOID) - input
The base address of the range of pages to be queried.

pcb (PULONG) - in/out
A pointer to a ULONG containing the size of the range of pages.

Input	This parameter points to a variable that contains the size, in bytes, of the range of pages to be queried. The initial value of the variable is rounded to include all pages addressed by the requested base address and size.
Output	This parameter points to a variable that contains the actual size, in bytes, of the queried range of pages.

pFlag (PULONG) - output
A pointer to a ULONG containing a set of attribute flags describing the type of allocation and access protection for the specified range of pages.

Allocation Type

PAG_COMMIT (0x00000010)
Pages within the specified region are committed.

PAG_SHARED (0x00002000)
Pages within the specified region are in a shared memory object. Otherwise, the pages are in a private memory object.

PAG_FREE (0x00004000)
Pages within the specified region are free.

PAG_BASE (0x00010000)
First page in the specified region is the first page in an allocated memory object.

Access Protection

PAG_READ (0x00000001)

Read access to the committed range of pages is allowed.

PAG_WRITE (0x00000002)
write access to the committed range of pages is allowed.

PAG_EXECUTE (0x00000004)
Execute access to the committed range of pages is allowed.

PAG_GUARD (0x00000008)
Access to the committed range of pages causes a "guard page entered" condition to be raised in the subject process.

ulrc (APIRET) - returns
Return Code.

DosQueryMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
487	ERROR_INVALID_ADDRESS

For a full list of error codes, see Errors.

DosQueryMem - Remarks

DosQueryMem provides the capability to determine the type and access protection of a range of pages within the virtual-address space of the subject process. This is the only memory-management function that accepts an address range that is not entirely contained within a previously allocated memory object.

The state of the first page within the region is determined, then subsequent entries in the virtual-address space of the process are scanned from the base address upward until either the entire range of pages has been scanned, a page with a nonmatching set of attributes is encountered, or the first page in an adjacent allocated memory object is encountered. The region attributes, the length of the range of pages with matching attributes, and an appropriate error code are returned.

If the entire requested range of pages does not have a matching set of attributes, then the returned *pcb* parameter value can be used to calculate the address and length of the range of pages that were not scanned.

Page scanning stops when the first page in an adjacent allocated memory object is encountered. This allows the calling application to determine the appearance of the virtual memory map, including object boundaries.

A region of pages that is neither committed nor free is considered reserved, that is, it is contained within an allocated memory object but has an access protection of "no access".

If the allocation type returned indicates that the pages are reserved, that is, neither PAG_COMMIT nor PAG_FREE is set, then the access protection returned is the same as was specified when the object was allocated in the address space of the requesting process.

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosQueryMem - Related Functions

Related Functions

- DosSetMem

DosQueryMem - Example Code

This example allocates, commits, queries, uses, and then frees read/write memory.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main (VOID)
{
   PVOID  MyObject    = NULL;      /* Pointer to memory object             */
   ULONG  ulObjSize   = 0;         /* Size of memory object (in bytes)     */
   PULONG  pulMemFlags = 0;        /* Allocation flags for the object      */
   PULONG  pulMemSize  = 0;        /* Size of memory region for DosSetMem  */
   APIRET rc          = NO_ERROR;  /* Return code                          */

   ulObjSize = 2000;          /* Will be rounded to a page boundary - 4096 */

   rc = DosAllocMem(&MyObject, /* Pointer to memory object pointer     */
                    ulObjSize,     /* Size of object to be allocated       */
                    PAG_WRITE );   /* Allocate memory as read/writeable    */
   if (rc != NO_ERROR) {
      printf("DosAllocMem error:  return code = %u\n",rc);
      return 1;
   }
       /* Object can't be used until it is COMMITTED.  Since this was
          not done at DosAllocMem time, do it now.                    */

   rc = DosSetMem(MyObject,                    /* Pointer to object      */
                  ulObjSize,                   /* Size of area to change */
                  PAG_DEFAULT | PAG_COMMIT );  /* Commit the object      */

   if (rc != NO_ERROR) {
      printf("DosSetMem error:  return code = %u\n",rc);
      rc = DosFreeMem(MyObject); /* If omitted, OS/2 frees it at termination */
      return 1;
   } else { printf("DosSetMem: complete\n"); }

   strcpy(MyObject, "The memory object has just been used.");

              /* Check COMMIT status of the memory object. */
   pulMemSize = ulObjSize;
   rc = DosQueryMem(MyObject, &pulMemSize, &pulMemFlags);
   if (rc == NO_ERROR) {
      if (pulMemFlags & PAG_COMMIT) {
         printf("   Page containing MyObject is now committed.\n");
      } else {
         printf("Error:  Page containing MyObject has not been committed.\n");
      } /* endif */
   } else {
      printf("DosQueryMem error: return code = %u\n",rc);
   } /* endif */

   rc = DosFreeMem(MyObject);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error:  return code = %u\n",rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosQueryMem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryMessageCP

DosQueryMessageCP - Syntax

Retrieves a message file list of code pages and language identifiers.

```
#define INCL_DOSMISC
#include <os2.h>

PCHAR     pb;           /*  Pointer to the caller's buffer area. */
ULONG     cb;           /*  The length, in bytes, of pb. */
PSZ       pszFilename;  /*  The file specification of the message file. */
PULONG    cbBuf;        /*  Pointer to the ULONG that receives the actual length, in bytes, of the returned data
APIRET    ulrc;         /*  Return Code. */

ulrc = DosQueryMessageCP(pb, cb, pszFilename,
         cbBuf);
```

DosQueryMessageCP Parameter - pb

pb (PCHAR) - output
Pointer to the caller's buffer area.

This buffer area is where the system returns the requested message file list of code pages and language identifiers.

DosQueryMessageCP Parameter - cb

cb (ULONG) - input
The length, in bytes, of *pb* .

DosQueryMessageCP Parameter - pszFilename

pszFilename (PSZ) - input
The file specification of the message file.

The drive designation and path are optional. This specifies a file that was previously prepared by the MKMSGF utility program.

DosQueryMessageCP Parameter - cbBuf

cbBuf (PULONG) - output
Pointer to the ULONG that receives the actual length, in bytes, of the returned data.

DosQueryMessageCP Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryMessageCP returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
206	ERROR_FILENAME_EXCED_RANGE
318	ERROR_MR_UN_ACC_MSGF
319	ERROR_MR_INV_MSGF_FORMAT
321	ERROR_MR_UN_PERFORM

For a full list of error codes, see Errors.

DosQueryMessageCP - Parameters

pb (PCHAR) - output
Pointer to the caller's buffer area.

This buffer area is where the system returns the requested message file list of code pages and language identifiers.

cb (ULONG) - input
The length, in bytes, of *pb* .

pszFilename (PSZ) - input
The file specification of the message file.

The drive designation and path are optional. This specifies a file that was previously prepared by the MKMSGF utility program.

cbBuf (PULONG) - output
Pointer to the ULONG that receives the actual length, in bytes, of the returned data.

ulrc (APIRET) - returns
Return Code.

DosQueryMessageCP returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
206	ERROR_FILENAME_EXCED_RANGE
318	ERROR_MR_UN_ACC_MSGF
319	ERROR_MR_INV_MSGF_FORMAT
321	ERROR_MR_UN_PERFORM

For a full list of error codes, see Errors.

DosQueryMessageCP - Remarks

DosQueryMessageCP retrieves the message file list of code pages and language identifiers.

The system returns the requested message file list of code pages and language identifiers in the caller's buffer (*pb*). It has the following format:

Length	Field
USHORT	Code page count (N)
USHORT	Code page identifier. This field occurs N times, once per code page.
ULONG	Language identifier

This data structure is repeated for each subfile within the specified message file.

The code page identifier can have the values specified in COUNTRYCODE.

The language identifier is a ULONG. The low-order word identifies a language family, and the high-order word identifies a specific version of that language (a sublanguage).

The language and sublanguage identifier values 0 through 255 are reserved for system use. The values 256 through 511 are reserved for application use.

The MKMSGF utility program performs range checking on the language and sublanguage identifier values. The value 0 means a null or unspecified language or sublanguage. Only the values defined in the following table are valid below 256. Any values from 256 through 511 are valid. Any identifier value greater than 511 is invalid.

The following table shows the valid language and sublanguage identifier values. Column 1 is the language family identifier, and column 2 is the sublanguage identifier. Column 3 shows the language, and column 4 shows the principal country for this language.

Family	Subl.	Language	Principal country
0	0	null	null
1	1	Arabic	Arab countries
2	1	Bulgarian	Bulgaria
3	2	Spanish	Spain
3	3	Spanish Mexican	Mexico
4	1	Traditional Chinese	Republic of China
4	2	Simplified Chinese	People's Republic of China
5	1	Czech	Czechoslovakia
6	1	Danish	Denmark
7	1	German	Germany
7	2	Swiss German	Switzerland
8	1	Greek	Greece
9	1	U.K. English	United Kingdom
9	2	U.S. English	United States
10	1	Finnish	Finland

11	1	French	France
11	2	Belgian French	Belgium
11	3	Canadian French	Canada
11	4	Swiss French	Switzerland
12	1	Hebrew	Israel
13	1	Hungarian	Hungary
14	1	Icelandic	Iceland
15	1	Italian	Italy
15	2	Swiss Italian	Switzerland
16	1	Japanese	Japan
17	1	Korean	Korea
18	1	Dutch	Netherlands
18	2	Belgian Dutch	Belgium
19	1	Norwegian - Bokmal	Norway
19	2	Norwegian - Nynorsk	Norway
20	1	Polish	Poland
21	1	Portuguese	Portugal
22	2	Brazilian Portuguese	Brazil
23	1	Rhaeto-Romanic	Switzerland
24	1	Serbo-Croatian (Cyrillic)	Yugoslavia
24	2	Serbo-Croatian (Latin)	Yugoslavia
25	1	Slovakian	Czechoslovakia
26	1	Albanian	Albania
27	1	Swedish	Sweden
28	1	Thai	Thailand
29	1	Turkish	Turkey
30	1	Urdu	Pakistan
31	1	Russian	U.S.S.R.
32	1	Catalan	Spain

DosQueryMessageCP - Related Functions

Related Functions

- DosGetMessage
- DosInsertMessage
- DosPutMessage

DosQueryMessageCP - Example Code

This example queries the the number of code pages and the first code page and language identifier from the file "SAMPLE.MSG". The result of the query has the format specified be the _MYCPINFO structure.

```
#define INCL_DOSMISC    /* Miscellaneous values */
#define INCL_DOSERRORS  /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

typedef struct _MYCPINFO
 {  USHORT       cCP;          /* Number of code pages */
    USHORT       CP1;          /* First code page      */
    ULONG        dtype;
 } MYCPINFO;

UCHAR     uchBuffer[20] = {0};            /* Buffer for the returned list */
MYCPINFO  CPinfo         = {0};           /* Pointer to buffer */
ULONG     ulDataLen      = 0;             /* Length of data returned */
APIRET    rc             = NO_ERROR;      /* Return code */

 rc = DosQueryMessageCP( (PUCHAR) &CPinfo,
                         sizeof(MYCPINFO),
                         "SAMPLE.MSG",
                         &ulDataLen );

 if (rc != NO_ERROR) {
      printf("DosQueryMessageCP error: return code = %u\n", rc);
      return 1;
 } else {
      printf("SAMPLE.MSG:  %u Code Page(s) - Code Page 1 = %u.\n",
                                      CPinfo.cCP, CPinfo.CP1);
 } /* endif */

return NO_ERROR;
}
```

DosQueryMessageCP - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryModuleHandle

DosQueryModuleHandle - Syntax

Returns the handle of a dynamic link module that was previously loaded.

```
#define INCL_DOSMODULEMGR
```

```
#include <os2.h>

PSZ         pszModname;
PHMODULE    phmod;
APIRET      ulrc;        /*  Return Code. */

ulrc = DosQueryModuleHandle(pszModname, phmod);
```

DosQueryModuleHandle Parameter - pszModname

pszModname (PSZ) - input
The address of an ASCIIZ name string containing the dynamic link module name. The file-name extension used for dynamic link libraries is .DLL.

DosQueryModuleHandle Parameter - phmod

phmod (PHMODULE) - output
The address of a doubleword in which the handle for the dynamic link module is returned.

DosQueryModuleHandle Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryModuleHandle returns one of the following values:

0	NO_ERROR
123	ERROR_INVALID_NAME

For a full list of error codes, see Errors.

DosQueryModuleHandle - Parameters

pszModname (PSZ) - input
The address of an ASCIIZ name string containing the dynamic link module name. The file-name extension used for dynamic link libraries is .DLL.

phmod (PHMODULE) - output
The address of a doubleword in which the handle for the dynamic link module is returned.

ulrc (APIRET) - returns
Return Code.

DosQueryModuleHandle returns one of the following values:

0	NO_ERROR
123	ERROR_INVALID_NAME

For a full list of error codes, see Errors.

DosQueryModuleHandle - Remarks

DosQueryModuleHandle returns the handle of a dynamic link module that was previously loaded.

The module name must match the name of the module already loaded. Otherwise, an error code is returned. This is a way of testing whether a dynamic link module is already loaded.

DosQueryModuleHandle - Related Functions

Related Functions

- DosFreeModule
- DosLoadModule
- DosQueryModuleName

DosQueryModuleHandle - Example Code

This example attempts to obtain the handle of a dynamic link module. This allows the caller to test whether a given dynamic link module is currently loaded.

```
#define INCL_DOSMODULEMGR   /* Module Manager values */
#define INCL_DOSERRORS      /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
   PSZ     ModuleName    = "C:\\OS2\\DLL\\DISPLAY.DLL";  /* Module name   */
   HMODULE ModuleHandle  = NULLHANDLE;                   /* Module handle */
   APIRET  rc            = NO_ERROR;                     /* Return code   */

   rc = DosQueryModuleHandle(ModuleName,      /* Module to look for       */
                             &ModuleHandle);  /* Handle (returned)        */
   if (rc != NO_ERROR) {
      printf("DosQueryModuleHandle error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("Module handle = %u\n", ModuleHandle);
   }

   rc = DosQueryModuleName (ModuleHandle,   /* Module handle to query     */
                            256L,           /* Maximum length of result   */
                            ModuleName);    /* Module name returned       */

   if (rc != NO_ERROR) {
      printf("DosQueryModuleName error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("Module name = %s\n", ModuleName);
   }
   return NO_ERROR;
}
```

DosQueryModuleHandle - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryModuleName

DosQueryModuleName - Syntax

Returns the fully-qualified drive, path, file name, and extension associated with the referenced module handle.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

HMODULE    hmod;    /*  The handle of the dynamic link module that is being referenced. */
ULONG      cbName;  /*  The maximum length of the buffer, in bytes, where the name will be stored. */
PCHAR      pch;     /*  The address of the buffer where the file specification of the module are returned. */
APIRET     ulrc;    /*  Return Code. */

ulrc = DosQueryModuleName(hmod, cbName, pch);
```

DosQueryModuleName Parameter - hmod

hmod (HMODULE) - input
The handle of the dynamic link module that is being referenced.

This handle is provided in the DI register on entry to a module, or on the initialization entry to a dynamic link routine.

DosQueryModuleName Parameter - cbName

cbName (ULONG) - input
The maximum length of the buffer, in bytes, where the name will be stored.

DosQueryModuleName Parameter - pch

pch (PCHAR) - output
The address of the buffer where the file specification of the module are returned.

DosQueryModuleName Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryModuleName returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
24	ERROR_BAD_LENGTH

For a full list of error codes, see Errors.

DosQueryModuleName - Parameters

hmod (HMODULE) - input
The handle of the dynamic link module that is being referenced.

This handle is provided in the DI register on entry to a module, or on the initialization entry to a dynamic link routine.

cbName (ULONG) - input
The maximum length of the buffer, in bytes, where the name will be stored.

pch (PCHAR) - output
The address of the buffer where the file specification of the module are returned.

ulrc (APIRET) - returns
Return Code.

DosQueryModuleName returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
24	ERROR_BAD_LENGTH

For a full list of error codes, see Errors.

DosQueryModuleName - Remarks

DosQueryModuleName returns the fully qualified drive, path, file name, and extension associated with the referenced module handle.

If the buffer is not large enough, an error is returned.

DosQueryModuleName - Related Functions

Related Functions

- DosFreeModule
- DosLoadModule
- DosQueryModuleHandle

DosQueryModuleName - Example Code

This example attempts to obtain the handle of a dynamic link module named "DISPLAY.DLL," and then tests whether it is currently loaded or not.

```
#define INCL_DOSMODULEMGR   /* Module Manager values */
#define INCL_DOSERRORS      /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
   PSZ     ModuleName    = "C:\\OS2\\DLL\\DISPLAY.DLL";  /* Module name   */
   HMODULE ModuleHandle  = NULLHANDLE;                   /* Module handle */
   APIRET  rc            = NO_ERROR;                     /* Return code   */

   rc = DosQueryModuleHandle(ModuleName,      /* Module to look for       */
                             &ModuleHandle);  /* Handle (returned)        */
   if (rc != NO_ERROR) {
      printf("DosQueryModuleHandle error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("Module handle = %u\n", ModuleHandle);
   }

   rc = DosQueryModuleName (ModuleHandle,   /* Module handle to query     */
                            256L,           /* Maximum length of result   */
                            ModuleName);    /* Module name returned       */

   if (rc != NO_ERROR) {
      printf("DosQueryModuleName error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("Module name = %s\n", ModuleName);
   }
   return NO_ERROR;
}
```

DosQueryModuleName - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryMutexSem

DosQueryMutexSem - Syntax

Retrieves information about the owner of a mutex semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMTX      hmtx;       /*  The handle of the mutex semaphore to query. */
PID      *ppid;       /*  A pointer to the process ID of either the current owner of the mutex semaphore, or a pr
TID      *ptid;       /*  A pointer to the thread ID of either the current owner of the mutex semaphore, or a pro
PULONG    pulCount;   /*  A pointer to the request count for the semaphore. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosQueryMutexSem(hmtx, ppid, ptid,
         pulCount);
```

DosQueryMutexSem Parameter - hmtx

hmtx (HMTX) - input
The handle of the mutex semaphore to query.

DosQueryMutexSem Parameter - ppid

ppid (PID *) - output
A pointer to the process ID of either the current owner of the mutex semaphore, or a process that has ended without releasing the semaphore.

DosQueryMutexSem Parameter - ptid

ptid (TID *) - output
A pointer to the thread ID of either the current owner of the mutex semaphore, or a process that has ended without releasing the semaphore.

DosQueryMutexSem Parameter - pulCount

pulCount (PULONG) - output
A pointer to the request count for the semaphore.

The request count is the number of calls to DosRequestMutexSem, minus the number of calls to DosReleaseMutexSem, that have been made for the semaphore by the owning thread. If the semaphore is unowned, this value will be zero. If the owning thread has ended, the value will be the request count for the ended owner.

DosQueryMutexSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED

For a full list of error codes, see Errors.

DosQueryMutexSem - Parameters

hmtx (HMTX) - input
The handle of the mutex semaphore to query.

ppid (PID *) - output
A pointer to the process ID of either the current owner of the mutex semaphore, or a process that has ended without releasing the semaphore.

ptid (TID *) - output
A pointer to the thread ID of either the current owner of the mutex semaphore, or a process that has ended without releasing the semaphore.

pulCount (PULONG) - output
A pointer to the request count for the semaphore.

The request count is the number of calls to DosRequestMutexSem, minus the number of calls to DosReleaseMutexSem, that have been made for the semaphore by the owning thread. If the semaphore is unowned, this value will be zero. If the owning thread has ended, the value will be the request count for the ended owner.

ulrc (APIRET) - returns
Return Code.

DosQueryMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED

For a full list of error codes, see Errors.

DosQueryMutexSem - Remarks

DosQueryMutexSem returns the process identification (PID) and thread identification (TID) of a mutex semaphore's current owner, as well as the request count for the semaphore. The request count is the number of calls to DosRequestMutexSem, minus the number of calls to DosReleaseMutexSem, that have been made for the semaphore by the owning thread.

This function can be called by any thread in the process that created the semaphore. Threads in other processes can also call this function, but they must first gain access to the semaphore by calling DosOpenMutexSem.

DosQueryMutexSem - Related Functions

Related Functions

- DosCloseMutexSem
- DosCreateMutexSem
- DosOpenMutexSem
- DosReleaseMutexSem
- DosRequestMutexSem

DosQueryMutexSem - Example Code

This example shows how to create, open, query, request, release, and close a Mutual exclusion (Mutex) semaphore.

```
#define INCL_DOSSEMAPHORES      /* Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HMTX    hmtx     = NULLHANDLE; /* Mutex semaphore handle */
PID     pidOwner = 0;          /* PID of current mutex semaphore owner */
TID     tidOwner = 0;          /* TID of current mutex semaphore owner */
ULONG   ulCount  = 0;          /* Request count for the semaphore */
APIRET  rc       = NO_ERROR;   /* Return code */

   rc = DosCreateMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                          &hmtx, 0, FALSE);       /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }
       /* This would normally be done by another unit of work */
   rc = DosOpenMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                        &hmtx);                 /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosRequestMutexSem(hmtx,                     /* Handle of semaphore */
                           (ULONG) SEM_INDEFINITE_WAIT);  /* Timeout (none) */
   if (rc != NO_ERROR) {
      printf("DosRequestMutexSem error: return code = %u\n", rc);
      return 1;
   }
   rc = DosQueryMutexSem(hmtx,         /* Handle of semaphore */
                         &pidOwner,    /* Process ID of owner */
                         &tidOwner,    /* Thread ID of owner */
                         &ulCount);    /* Count */
   if (rc != NO_ERROR) {
      printf("DosQueryMutexSem error: return code = %u\n", rc);
      return 1;
   } else  {
      printf("Semaphore owned by PID %u, TID %u.", pidOwner, tidOwner);
      printf("  Request count is %u.\n", ulCount);
   } /* endif */

   rc = DosReleaseMutexSem(hmtx);        /* Relinquish ownership */
   if (rc != NO_ERROR) {
      printf("DosReleaseMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosCloseMutexSem(hmtx);          /* Close mutex semaphore */
   if (rc != NO_ERROR) {
      printf("DosCloseMutexSem error: return code = %u\n", rc);
      return 1;
   }

return NO_ERROR;
}
```

DosQueryMutexSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryMuxWaitSem

DosQueryMuxWaitSem - Syntax

Retrieves the semaphore records from a muxwait-semaphore list.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMUX          hmux;      /*  The handle of the muxwait semaphore to query. */
PULONG        pcSemRec;  /*  A pointer to the ULONG which contains the number of semaphore record entries. */
PSEMRECORD    pSemRec;   /*  A pointer to the semaphore record entries in the muxwait-semaphore list. */
PULONG        pflAttr;   /*  A pointer to the flAttr attribute flags that were passed by DosCreateMuxWaitSem. */
APIRET        ulrc;      /*  Return Code. */

ulrc = DosQueryMuxWaitSem(hmux, pcSemRec,
         pSemRec, pflAttr);
```

DosQueryMuxWaitSem Parameter - hmux

hmux (HMUX) - input
The handle of the muxwait semaphore to query.

DosQueryMuxWaitSem Parameter - pcSemRec

pcSemRec (PULONG) - in/out
A pointer to the ULONG which contains the number of semaphore record entries.

Input	A pointer to the maximum number of semaphore record entries that can be contained in the list pointed to by *pSemRec* .
Output	A pointer to the number of semaphore record entries returned in the list pointed to by *pSemRec* . If the list

pointed to by *pSemRec* is not large enough to hold all of the semaphore records in the specified muxwait semaphore, the system sets the value pointed to by *pSemRec* to the number of semaphore records in the specified muxwait semaphore, and sets the return code to ERROR_PARAM_TOO_SMALL.

DosQueryMuxWaitSem Parameter - pSemRec

pSemRec (PSEMRECORD) - output
A pointer to the semaphore record entries in the muxwait-semaphore list.

This is the list of event or mutex semaphores that must be posted or released in order for the muxwait semaphore to be cleared.

DosQueryMuxWaitSem Parameter - pflAttr

pflAttr (PULONG) - output
A pointer to the *flAttr* attribute flags that were passed by DosCreateMuxWaitSem.

Possible flags are shown in the list below:

1 DC_SEM_SHARED
The semaphore is shared.

2 DCMW_WAIT_ANY
The semaphore waits for any event semaphore in the muxwait-semaphore list to be posted, or for any mutex semaphore in the list to be released. When any one of the semaphores is cleared, the thread that is waiting on the muxwait semaphore can continue executing.

4 DCMW_WAIT_ALL
The semaphore waits for all of the event semaphores in the muxwait list to be posted, or for all of the mutex semaphores in the list to be released. When all of the semaphores are cleared, the thread that is waiting on the muxwait semaphore can continue executing.

DosQueryMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED
289	ERROR_PARAM_TOO_SMALL

For a full list of error codes, see Errors.

DosQueryMuxWaitSem - Parameters

hmux (HMUX) - input
The handle of the muxwait semaphore to query.

pcSemRec (PULONG) - in/out
A pointer to the ULONG which contains the number of semaphore record entries.

Input	A pointer to the maximum number of semaphore record entries that can be contained in the list pointed to by *pSemRec* .
Output	A pointer to the number of semaphore record entries returned in the list pointed to by *pSemRec* . If the list pointed to by *pSemRec* is not large enough to hold all of the semaphore records in the specified muxwait semaphore, the system sets the value pointed to by *pSemRec* to the number of semaphore records in the specified muxwait semaphore, and sets the return code to ERROR_PARAM_TOO_SMALL.

pSemRec (PSEMRECORD) - output
A pointer to the semaphore record entries in the muxwait-semaphore list.

This is the list of event or mutex semaphores that must be posted or released in order for the muxwait semaphore to be cleared.

pflAttr (PULONG) - output
A pointer to the *flAttr* attribute flags that were passed by DosCreateMuxWaitSem.

Possible flags are shown in the list below:

1	DC_SEM_SHARED The semaphore is shared.
2	DCMW_WAIT_ANY The semaphore waits for any event semaphore in the muxwait-semaphore list to be posted, or for any mutex semaphore in the list to be released. When any one of the semaphores is cleared, the thread that is waiting on the muxwait semaphore can continue executing.
4	DCMW_WAIT_ALL The semaphore waits for all of the event semaphores in the muxwait list to be posted, or for all of the mutex semaphores in the list to be released. When all of the semaphores are cleared, the thread that is waiting on the muxwait semaphore can continue executing.

ulrc (APIRET) - returns
Return Code.

DosQueryMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
105	ERROR_SEM_OWNER_DIED
289	ERROR_PARAM_TOO_SMALL

For a full list of error codes, see Errors.

DosQueryMuxWaitSem - Remarks

DosQueryMuxWaitSem retrieves the semaphore records from a muxwait-semaphore list.

The process must have previously opened the muxwait semaphore by issuing DosCreateMuxWaitSem or DosOpenMuxWaitSem. If the muxwait semaphore does not exist, then the system returns the ERROR_INVALID_HANDLE return code to the caller.

The value that *pcSemRec* points to on input must be the maximum number of semaphore record entries that can be contained in the list pointed to by The value that *pSemRec* For example, if the list pointed to by *pSemRec* can contain ten semaphore record entries, then you should set the input value pointed to by *pcSemRec* to ten before issuing DosQueryMuxWaitSem.

If the list pointed to by *pSemRec* is not large enough to hold all of the semaphore records in the specified muxwait semaphore, then the system sets the value pointed to by *pcSemRec* to the number of semaphore records in the specified muxwait semaphore, and sets the return code to ERROR_PARAM_TOO_SMALL. This allows you to issue DosQueryMuxWaitSem again, with the correct amount of memory for the muxwait-semaphore list.

The system returns the ERROR_SEM_OWNER_DIED return code if any of the mutex semaphores in the muxwait semaphore have been placed into the owner-died state. This means that a thread ended while it owned at least one mutex semaphore, and at least one mutex semaphore is part of the muxwait semaphore. It also means that the mutex semaphore has not yet been removed by DosCloseMutexSem.

When the system returns the ERROR_SEM_OWNER_DIED return code, you should issue DosQueryMutexSem for each mutex semaphore in the muxwait-semaphore list to determine which ones are invalid. For each mutex semaphore that results in the ERROR_SEM_OWNER_DIED return code from DosQueryMutexSem, issue DosCloseMutexSem to close the mutex semaphore. Semaphore handles may be used again, so the mutex semaphores that are closed *must* be deleted from the muxwait semaphore.

DosQueryMuxWaitSem - Related Functions

Related Functions

- DosAddMuxWaitSem
- DosCloseMuxWaitSem
- DosCreateMuxWaitSem
- DosDeleteMuxWaitSem
- DosOpenMuxWaitSem
- DosWaitMuxWaitSem

DosQueryMuxWaitSem - Example Code

This example creates a muxwait semaphore with two event semaphores in its list, queries the number of entries, and displays it.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX       hmuxHandAny  = NULLHANDLE;   /* Muxwaithandle */
 HEV        hev[2]       = {0};          /* Event semaphores */
 SEMRECORD  apsr[2]      = {{0}};        /* Semaphore records */
 SEMRECORD  semrecQuery[2] = {{0}};        /* Pointer from query */
 ULONG      cQueryRec    = 2;            /* Number of records found by query */
 ULONG      fQueryFlags  = 0;            /* Attribute flags returned by query */
 APIRET     rc           = NO_ERROR;     /* Return code */
 ULONG      ulLoop       = 0;            /* Loop count */
 ULONG      ulSem        = 0;

for (ulLoop = 0; ulLoop < 2; ulLoop++) {
    rc = DosCreateEventSem((PSZ) NULL, &hev[ulLoop], 0, FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
    apsr[ulLoop].hsemCur = (HSEM) hev[ulLoop],
    apsr[ulLoop].ulUser = 0;
} /* endfor */

rc = DosCreateMuxWaitSem((PSZ) NULL, &hmuxHandAny, 2L, apsr, DCMW_WAIT_ANY);
if (rc != NO_ERROR) {
   printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
   return 1;
}
           /* Query information about the MuxWait semaphore */

rc = DosQueryMuxWaitSem(hmuxHandAny,    /* Semaphore handle */
                        &cQueryRec,     /* Number of records */
                        (PSEMRECORD) semrecQuery,  /* Pointer to Semrecords */
                        &fQueryFlags);  /* Flags returned */
if (rc != NO_ERROR) {
  printf("DosQueryMuxWaitSem error:  return code = %u\n", rc);
  return 1;
} else {
  printf("DosQueryMuxWaitSem found %u semaphore records\n", cQueryRec);
} /* endif */
```

```
return NO_ERROR;
}
```

DosQueryMuxWaitSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryNPHState

DosQueryNPHState - Syntax

Returns information about a named-pipe handle.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE     hpipe;   /*  The named-pipe handle to query. */
PULONG    pState;  /*  A pointer to the named-pipe handle state. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosQueryNPHState(hpipe, pState);
```

DosQueryNPHState Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to query.

The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen.

DosQueryNPHState Parameter - pState

pState (PULONG) - output
A pointer to the named-pipe handle state.

This parameter contains the following bit fields:

Bit	Description
31-16	Reserved.
15	Blocking mode. Blocking mode is defined as either "blocking" or "nonblocking," as follows: 0 NP_WAIT (0x0000) Blocking mode: DosRead and DosWrite block if no data is available. 1 NP_NOWAIT (0x8000) Nonblocking mode: DosRead and DosWrite return immediately if no data is available. DosRead normally blocks until at least partial data can be returned. DosWrite blocks by default until all of the requested bytes have been written. Nonblocking mode changes this behavior as follows: DosRead returns immediately with a value of zero for *pcbActual* if no data is available. DosWrite returns immediately with a value of zero for *pcbActual* if there is not enough buffer space available in the pipe; otherwise, the entire data area is transferred.
14	Specifies whether the handle is for the server or client end of the pipe, as follows: 0 NP_END_CLIENT (0x0000) The handle is for the client end of the pipe. 1 NP_END_SERVER (0x4000) The handle is for the server end of the pipe.
13-12	Reserved.
11-10	Type of named pipe. The pipe type is defined as follows: 00 NP_TYPE_BYTE (0x0000) The pipe is a byte pipe; that is, data is written to the pipe as an undifferentiated stream of bytes. 01 NP_TYPE_MESSAGE (0x0400) The pipe is a message pipe; that is, data is written to the pipe as messages. The system records the length of each message in the first two bytes of the message, which are called the *message header* .
9-8	Read mode. The read mode is defined as follows: 00 NP_READMODE_BYTE (0x0000) Byte-read mode: Read the pipe as a byte stream. 01 NP_READMODE_MESSAGE (0x0100) Message-read mode: Read the pipe as a message stream.
7-0	Instance count: When the first instance of a named pipe is created, this field specifies how many instances (including the first instance) can be created. Possible values are: Value — Definition 1 — This is the only instance permitted (the pipe is unique). 1 < value < 255 — The number of instances is limited to the value specified. -1 — NP_UNLIMITED_INSTANCES (0x00FF) The number of instances is unlimited. 0 — Reserved value.

DosQueryNPHState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryNPHState returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
230	ERROR_BAD_PIPE
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosQueryNPHState - Parameters

hpipe (HPIPE) - input
The named-pipe handle to query.

The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen.

pState (PULONG) - output
A pointer to the named-pipe handle state.

This parameter contains the following bit fields:

Bit	Description
31-16	Reserved.
15	Blocking mode. Blocking mode is defined as either "blocking" or "nonblocking," as follows: 0 NP_WAIT (0x0000) Blocking mode: DosRead and DosWrite block if no data is available. 1 NP_NOWAIT (0x8000) Nonblocking mode: DosRead and DosWrite return immediately if no data is available. DosRead normally blocks until at least partial data can be returned. DosWrite blocks by default until all of the requested bytes have been written. Nonblocking mode changes this behavior as follows: DosRead returns immediately with a value of zero for *pcbActual* if no data is available. DosWrite returns immediately with a value of zero for *pcbActual* if there is not enough buffer space available in the pipe; otherwise, the entire data area is transferred.
14	Specifies whether the handle is for the server or client end of the pipe, as follows: 0 NP_END_CLIENT (0x0000) The handle is for the client end of the pipe. 1 NP_END_SERVER (0x4000) The handle is for the server end of the pipe.
13-12	Reserved.
11-10	Type of named pipe. The pipe type is defined as follows: 00 NP_TYPE_BYTE (0x0000) The pipe is a byte pipe; that is, data is written to the pipe as an undifferentiated stream of bytes. 01 NP_TYPE_MESSAGE (0x0400) The pipe is a message pipe; that is, data is written to the pipe as messages. The system records the length of each message in the first two bytes of the message, which are called the *message header* .
9-8	Read mode. The read mode is defined as follows:

	00	NP_READMODE_BYTE (0x0000) Byte-read mode: Read the pipe as a byte stream.
	01	NP_READMODE_MESSAGE (0x0100) Message-read mode: Read the pipe as a message stream.
7-0	Instance count: When the first instance of a named pipe is created, this field specifies how many instances (including the first instance) can be created. Possible values are:	

Value	Definition
1	This is the only instance permitted (the pipe is unique).
1 < value < 255	The number of instances is limited to the value specified.
-1	NP_UNLIMITED_INSTANCES (0x00FF) The number of instances is unlimited.
0	Reserved value.

ulrc (APIRET) - returns
Return Code.

DosQueryNPHState returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
230	ERROR_BAD_PIPE
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosQueryNPHState - Remarks

DosQueryNPHState returns the following information about a pipe handle and the attributes of the pipe:

- The end of the pipe that the handle is for (server or client end)
- The pipe type (byte pipe or message pipe)
- The instance count
- The blocking mode (blocking or nonblocking)
- The read mode (byte-read mode or message-read mode).

The values for the pipe type and instance count cannot be changed, so they are always the same as those that were specified when the pipe was created with DosCreateNPipe. The information returned for blocking mode and read mode, however, can come from different sources:

- If the handle is for the server end of the pipe, then the blocking mode and the read mode were set with DosCreateNPipe, but may have been reset with DosSetNPHState.
- If the handle is for the client end of the pipe, then the blocking mode and the read mode were set to "blocking" and "byte-read" by the system when the client issued DosOpen. However, they may have been reset with DosSetNPHState.

DosQueryNPHState - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle

- DosOpen
- DosPeekNPipe
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosQueryNPHState - Example Code

This example handles the client side of a pipe. It opens an existing named pipe, sets and queries the pipe handle state, sends a message to the host, reads the host reply, and finally closes the pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   ULONG    PipeState             = 0;
   ULONG    HandType              = 0;
   ULONG    FlagWord              = 0;
   ULONG    BytesRead             = 0;

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   } else printf("Connected to HOST\n");

   rc = DosSetNPHState(PipeHandle, NP_WAIT);
   if (rc != NO_ERROR) {
      printf("DosSetNPHState error: error code = %u\n", rc);
      return 1;
   }

   rc = DosQueryNPHState(PipeHandle, &PipeState);
   if (rc != NO_ERROR) {
      printf("DosQueryNPHState error: error code = %u\n", rc);
      return 1;
   } else {
      printf("Pipe handle state is: %x\n", PipeState);
   }  /* endif */

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);  /* Force above printf prompt to display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }
```

```
   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST was: %s\n\n", message);

   rc = DosClose(PipeHandle);
   /* Should check if (RC == NO_ERROR)  here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosQueryNPHState - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryNPipeInfo

DosQueryNPipeInfo - Syntax

Returns information about a named pipe.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE     hpipe;      /*  The named-pipe handle to query. */
ULONG     infolevel;  /*  Level of the required pipe information. */
PVOID     pBuf;       /*  A pointer to the storage area in which the requested level of named-pipe information i
ULONG     cbBuf;      /*  The length, in bytes, of pBuf. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosQueryNPipeInfo(hpipe, infolevel,
         pBuf, cbBuf);
```

DosQueryNPipeInfo Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to query.

(The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen).

DosQueryNPipeInfo Parameter - infolevel

infolevel (ULONG) - input
Level of the required pipe information.

Only levels 1 and 2 are supported.

DosQueryNPipeInfo Parameter - pBuf

pBuf (PVOID) - output
A pointer to the storage area in which the requested level of named-pipe information is returned.

Level 1 File Information
Information about the pipe itself is returned in the form of a PIPEINFO structure.

Level 2 File Information
For LAN Manager, LAN Server, and other LAN-based named pipe handles, the buffer will contain the unique 4-byte identifier of the client. Otherwise, this field returns NULL.

DosQueryNPipeInfo Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of *pBuf*.

DosQueryNPipeInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryNPipeInfo returns one of the following values:

0	NO_ERROR
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
230	ERROR_BAD_PIPE

For a full list of error codes, see Errors.

DosQueryNPipeInfo - Parameters

hpipe (HPIPE) - input
The named-pipe handle to query.

(The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen).

infolevel (ULONG) - input
Level of the required pipe information.

Only levels 1 and 2 are supported.

pBuf (PVOID) - output
A pointer to the storage area in which the requested level of named-pipe information is returned.

Level 1 File Information
Information about the pipe itself is returned in the form of a PIPEINFO structure.

Level 2 File Information
For LAN Manager, LAN Server, and other LAN-based named pipe handles, the buffer will contain the unique 4-byte identifier of the client. Otherwise, this field returns NULL.

cbBuf (ULONG) - input
The length, in bytes, of *pBuf*.

ulrc (APIRET) - returns
Return Code.

DosQueryNPipeInfo returns one of the following values:

0	NO_ERROR
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
230	ERROR_BAD_PIPE

For a full list of error codes, see Errors.

DosQueryNPipeInfo - Remarks

DosQueryNPipeInfo returns all of the level-1 or level-2 information about a named pipe that will fit in the *pBuf* storage area.

If the length of the pipe name is greater than 255 bytes, then a length of 0 is returned in the *namelength* field; however, the full ASCIIZ name is still returned in the *pipename* field.

If there is more information than will fit in *pBuf* ERROR_BUFFER_OVERFLOW is returned.

DosQueryNPipeInfo - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState

- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosQueryNPipeInfo - Example Code

This example handles the client side of a pipe. It opens an existing named pipe, sends a message to the host, reads the host reply, queries and displays the pipe name, and finally closes the pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(VOID) {
   APIRET   rc                     = NO_ERROR;   /* Return code */
   CHAR     message[256]           = "";         /* Message buffer */
   HFILE    PipeHandle             = NULLHANDLE; /* Pipe handle */
   PIPEINFO PipeBuffer[4]          = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]            = {0};
   ULONG    bytes                  = 0;
   ULONG    Action                 = 0;
   PIPESEMSTATE infobuf[3]         = {{0}};
   int i = 0;

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to Pipe.\n");

   printf("Enter message to send to HOST: ");

   fflush(NULL);   /* Force printf to display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }

   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST: %s\n\n", message);

   rc = DosQueryNPipeInfo(PipeHandle, 1L, &PipeBuffer, sizeof(PIPEINFO)*4);
   if (rc == NO_ERROR) {
     printf("The pipe's name is %s\n", PipeBuffer[0].szName);
   }

   rc = DosClose(PipeHandle);
   /* Should verify that (rc != NO_ERROR) here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosQueryNPipeInfo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryNPipeSemState

DosQueryNPipeSemState - Syntax

Returns information about local named pipes that are attached to a semaphore.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HSEM            hsem;   /*  The handle of a shared event or muxwait semaphore that was previously attached to o
PPIPESEMSTATE   pnpss;  /*  A pointer to a buffer containing a record for each named pipe that is attached to t
ULONG           cbBuf;  /*  The size, in bytes, of pnpss. */
APIRET          ulrc;   /*  Return Code. */

ulrc = DosQueryNPipeSemState(hsem, pnpss,
         cbBuf);
```

DosQueryNPipeSemState Parameter - hsem

hsem (HSEM) - input
The handle of a shared event or muxwait semaphore that was previously attached to one or more named pipes with DosSetNPipeSem.

DosQueryNPipeSemState Parameter - pnpss

pnpss (PPIPESEMSTATE) - output
A pointer to a buffer containing a record for each named pipe that is attached to the semaphore.

DosQueryNPipeSemState Parameter - cbBuf

cbBuf (ULONG) - input
The size, in bytes, of *pnpss* .

DosQueryNPipeSemState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryNPipeSemState returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

For a full list of error codes, see Errors.

DosQueryNPipeSemState - Parameters

hsem (HSEM) - input
The handle of a shared event or muxwait semaphore that was previously attached to one or more named pipes with DosSetNPipeSem.

pnpss (PPIPESEMSTATE) - output
A pointer to a buffer containing a record for each named pipe that is attached to the semaphore.

cbBuf (ULONG) - input
The size, in bytes, of *pnpss* .

ulrc (APIRET) - returns
Return Code.

DosQueryNPipeSemState returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

For a full list of error codes, see Errors.

DosQueryNPipeSemState - Remarks

DosQueryNPipeSemState returns information about the status of local named pipes that are attached to a shared event or multiple-wait (muxwait) semaphore. (Event semaphores are attached to local named pipes by calling DosSetNPipeSem.)

A record is returned for each local named pipe that is attached to the specified semaphore and whose state is either closed or allows blocking-mode input and output to be done. However, there is no guarantee that the records in the buffer refer only to named pipes that were opened by the process making this call. If the same semaphore has been attached to different named pipes by multiple processes, information about named pipes that are not accessible to the caller can be returned. For this reason, communicating processes should have a convention for key values to help identify the named pipes of interest. (A key value is specified when DosSetNPipeSem is called to attach the semaphore to a named pipe.)

If a process wants data in the buffer to refer only to its own named pipes, it must use a private event semaphore.

DosQueryNPipeSemState - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosQueryNPipeSemState - Example Code

This example handles the client side of a pipe. It opens an existing named pipe and event semaphore, sends a message to the host, displays the state of the pipe semaphore, reads the host reply, and finally closes the event semaphore and pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   HEV      hev                   = NULLHANDLE; /* Event semaphore handle */
   PIPEINFO PipeBuffer[4]         = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   int      i                     = 0;
   PIPESEMSTATE infobuf[3]        = {{0}};

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to Pipe.\n");

   rc = DosOpenEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev);
   if (rc != NO_ERROR) {
      printf("DosOpenEventSem error: error code = %u\n", rc);
      return 1;
   }

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);  /* Flush printf buffer */
```

```
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }

   rc = DosQueryNPipeSemState((HSEM) hev, infobuf, sizeof(PIPESEMSTATE)*3);
   if (rc != NO_ERROR) {
     printf("DosQueryNPipeSemState error: return code = %u\n",rc);
     return 1;
   }

   printf("Status  Flag   Key   Avail\n------ ------ ------ ------\n");
   for (i=0; i<3; i++)
     printf("%6u %6u %6u %6u\n", infobuf[i].fStatus,
            infobuf[i].fFlag, infobuf[i].usKey, infobuf[i].usAvail);

   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST: %s\n\n", message);

   rc = DosCloseEventSem(hev);
   /* Should check if (rc != NO_ERROR) here... */

   rc = DosClose(PipeHandle);
   /* Should check if (rc != NO_ERROR) here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosQueryNPipeSemState - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryPathInfo

DosQueryPathInfo - Syntax

Gets file information for a file or subdirectory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszPathName;  /*  Address of the ASCIIZ file specification of the file or subdirectory. */
```

```
ULONG     ulInfoLevel;  /*  The level of path information required. */
PVOID     pInfoBuf;     /*  Address of the storage area containing the requested level of path information. */
ULONG     cbInfoBuf;    /*  The length, in bytes, of pInfoBuf. */
APIRET    ulrc;         /*  Return Code. */

ulrc = DosQueryPathInfo(pszPathName, ulInfoLevel,
         pInfoBuf, cbInfoBuf);
```

DosQueryPathInfo Parameter - pszPathName

pszPathName (PSZ) - input
Address of the ASCIIZ file specification of the file or subdirectory.

Global file-name characters can be used in the name only for level 5 file information.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

DosQueryPathInfo Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
The level of path information required.

A value of 1, 2, 3, or 5 can be specified, as follows:

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information
3	FIL_QUERYEASFROMLIST Level 3 file information
5	FIL_QUERYFULLNAME Level 5 file information

Level 4 is reserved.

The structures described in *pInfoBuf* indicate the information returned for each of these levels.

DosQueryPathInfo Parameter - pInfoBuf

pInfoBuf (PVOID) - output
Address of the storage area containing the requested level of path information.

Path information, where applicable, is based on the most recent DosClose, DosResetBuffer, DosSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure, in which path information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG On output, this field contains the size, in bytes, of the file's entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file's entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)
This is a subset of the EA information of the file.

Input	*ulInfoLevel* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 that defines the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be zero. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfoBuf* is unchanged. If an error occurs, *oError* points to the GEA2 entry that caused the error. The buffer pointed to by *fpFEA2List* is filled in with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size, in bytes, of the file's entire EA set on disk, even though only a subset of attributes was requested. The size of the buffer required to hold the EA information is less than or equal to twice the size of the file's entire EA set on disk.

Level 5 File Information (*ulInfoLevel* == FIL_QUERYFULLNAME)
Level 5 returns the fully qualified ASCIIZ name of *pszPathName* in *pInfoBuf*. *pszPathName* may contain global file-name characters.

DosQueryPathInfo Parameter - cbInfoBuf

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

DosQueryPathInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryPathInfo returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
32	ERROR_SHARING_VIOLATION
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosQueryPathInfo - Parameters

pszPathName (PSZ) - input
Address of the ASCIIZ file specification of the file or subdirectory.

Global file-name characters can be used in the name only for level 5 file information.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

ulInfoLevel (ULONG) - input
The level of path information required.

A value of 1, 2, 3, or 5 can be specified, as follows:

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information
3	FIL_QUERYEASFROMLIST Level 3 file information
5	FIL_QUERYFULLNAME Level 5 file information

Level 4 is reserved.

The structures described in *pInfoBuf* indicate the information returned for each of these levels.

pInfoBuf (PVOID) - output
Address of the storage area containing the requested level of path information.

Path information, where applicable, is based on the most recent DosClose, DosResetBuffer, DosSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure, in which path information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG On output, this field contains the size, in bytes, of the file's entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file's entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)
This is a subset of the EA information of the file.

Input	*ulInfoLevel* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 that defines the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be zero. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfoBuf* is unchanged. If an error occurs, *oError* points to the GEA2 entry that caused the error. The buffer pointed to by *fpFEA2List* is filled in with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size, in bytes, of the file's entire EA set on disk, even though only a subset of attributes was requested. The size of the buffer required to hold the EA information is less than or equal to twice the size of the file's entire EA set on disk.

Level 5 File Information (*ulInfoLevel* == FIL_QUERYFULLNAME)
Level 5 returns the fully qualified ASCIIZ name of *pszPathName* in *pInfoBuf*. *pszPathName* may contain global file-name characters.

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

ulrc (APIRET) - returns
Return Code.

DosQueryPathInfo returns one of the following values:

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
32	ERROR_SHARING_VIOLATION
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosQueryPathInfo - Remarks

In the FAT file system, only date and time information contained in Level 1 file information can be modified. Zero is returned for the creation and access dates and times.

For DosQueryPathInfo to return information contained in any of the file information levels, the file object must be opened for read access, with a deny-write sharing mode specified for access by other processes. Thus, if the file object is already accessed by another process that holds conflicting sharing and access rights, a call to DosQueryPathInfo fails.

DosQueryPathInfo - Related Functions

Related Functions

- DosClose
- DosCreateDir
- DosEnumAttribute
- DosOpen
- DosQueryFileInfo
- DosResetBuffer
- DosSetFileInfo
- DosSetPathInfo

DosQueryPathInfo - Example Code

The first example obtains information about the file "STARTUP.CMD." The second example obtains information about the directory "SYSTEM."

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR        uchFileName[80] = "C:\\STARTUP.CMD";  /* File to manipulate    */
FILESTATUS3  fsts3ConfigInfo = {{0}};       /* Buffer for file information */
ULONG        ulBufSize     = sizeof(FILESTATUS3);  /* Size of above buffer */
APIRET       rc            = NO_ERROR;      /* Return code                 */

   rc = DosQueryPathInfo(uchFileName,   /* Path and name of file          */
                         FIL_STANDARD,  /* Request standard (Level 1) info */
                         &fsts3ConfigInfo, /* Buffer for file information  */
                         ulBufSize);    /* Size of buffer                 */
   if (rc != NO_ERROR) {
       printf("DosQueryPathInfo error: return code = %u\n", rc);
       return 1;
   }
```

```
    printf("%s ---  File size: %u bytes\n",uchFileName, fsts3ConfigInfo.cbFile);
    printf("Last updated: %d/%d/%d; %d:%2.2d\n",
            fsts3ConfigInfo.fdateLastWrite.month,        /* Month            */
            fsts3ConfigInfo.fdateLastWrite.day,          /* Day              */
            (fsts3ConfigInfo.fdateLastWrite.year+1980L), /* Years since 1980 */
            fsts3ConfigInfo.ftimeLastWrite.hours,        /* Hours            */
            fsts3ConfigInfo.ftimeLastWrite.minutes);     /* Minutes          */

  return NO_ERROR;
}



 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>

 int main(VOID) {
 UCHAR        uchPathName[255] = "C:\\OS2\\SYSTEM"; /* Path of interest     */
 FILESTATUS3  fsts3ConfigInfo = {{0}};       /* Buffer for path information */
 ULONG        ulBufSize     = sizeof(FILESTATUS3);  /* Size of above buffer */
 APIRET       rc            = NO_ERROR;      /* Return code                 */

    rc = DosQueryPathInfo(uchPathName,   /* Name of path                    */
                          FIL_STANDARD,  /* Request standard (Level 1) info */
                          &fsts3ConfigInfo, /* Buffer for information       */
                          ulBufSize);       /* Size of buffer              */
    if (rc != NO_ERROR) {
        printf("DosQueryPathInfo error: return code = %u\n", rc);
        return 1;
    }

    printf("Information for subdirectory: %s:\n",uchPathName);
    printf("Last updated: %d/%d/%d; %d:%2.2d\n",
            fsts3ConfigInfo.fdateLastWrite.month,        /* Month            */
            fsts3ConfigInfo.fdateLastWrite.day,          /* Day              */
            (fsts3ConfigInfo.fdateLastWrite.year+1980L), /* Years since 1980 */
            fsts3ConfigInfo.ftimeLastWrite.hours,        /* Hours            */
            fsts3ConfigInfo.ftimeLastWrite.minutes);     /* Minutes          */

  return NO_ERROR;
}
```

DosQueryPathInfo - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryProcAddr

DosQueryProcAddr - Syntax

Returns the address of the specified procedure within a dynamic link module.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

HMODULE    hmod;     /*  The handle of the dynamic link module that contains the procedure. */
ULONG      ordinal;  /*  The ordinal number of the procedure whose address is desired. */
PSZ        pszName;  /*  The address of an ASCIIZ name string that contains the procedure name that is being ref
PFN       *ppfn;     /*  A pointer to a PFN in which the procedure address is returned. */
APIRET     ulrc;     /*  Return Code. */

ulrc = DosQueryProcAddr(hmod, ordinal, pszName,
         ppfn);
```

DosQueryProcAddr Parameter - hmod

hmod (HMODULE) - input
The handle of the dynamic link module that contains the procedure.

DosQueryProcAddr Parameter - ordinal

ordinal (ULONG) - input
The ordinal number of the procedure whose address is desired.

If the ordinal number is nonzero, *pszName* is ignored. It must be less or equal to 65 533.

DosQueryProcAddr Parameter - pszName

pszName (PSZ) - input
The address of an ASCIIZ name string that contains the procedure name that is being referenced.

Calls to DosQueryProcAddr for entries within the DOSCALLS module are supported for ordinal references only. References to the DOSCALLS module by name strings are not supported, and will return an error. Dynamic link ordinal numbers for DOSCALLS routines are resolved by linking with OS2386.LIB.

DosQueryProcAddr Parameter - ppfn

ppfn (PFN *) - output
A pointer to a PFN in which the procedure address is returned.

DosQueryProcAddr Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryProcAddr returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
123	ERROR_INVALID_NAME
182	ERROR_INVALID_ORDINAL
65079	ERROR_ENTRY_IS_CALLGATE (this error code is not valid in OS/2 Warp PowerPC Edition)

For a full list of error codes, see Errors.

DosQueryProcAddr - Parameters

hmod (HMODULE) - input
The handle of the dynamic link module that contains the procedure.

ordinal (ULONG) - input
The ordinal number of the procedure whose address is desired.

If the ordinal number is nonzero, *pszName* is ignored. It must be less or equal to 65 533.

pszName (PSZ) - input
The address of an ASCIIZ name string that contains the procedure name that is being referenced.

Calls to DosQueryProcAddr for entries within the DOSCALLS module are supported for ordinal references only. References to the DOSCALLS module by name strings are not supported, and will return an error. Dynamic link ordinal numbers for DOSCALLS routines are resolved by linking with OS2386.LIB.

ppfn (PFN *) - output
A pointer to a PFN in which the procedure address is returned.

ulrc (APIRET) - returns
Return Code.

DosQueryProcAddr returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
123	ERROR_INVALID_NAME
182	ERROR_INVALID_ORDINAL
65079	ERROR_ENTRY_IS_CALLGATE (this error code is not valid in OS/2 Warp PowerPC Edition)

For a full list of error codes, see Errors.

DosQueryProcAddr - Remarks

DosQueryProcAddr returns the address of the specified procedure within a dynamic link module.

If you receive return code ERROR_INVALID_HANDLE, issue DosLoadModule and repeat this call.

If you issue DosQueryProcAddr to obtain the address of an entry point that may only be accessed via a call gate, you receive the return code ERROR_ENTRY_IS_CALLGATE.

DosQueryProcAddr - Related Functions

Related Functions

- DosFreeModule
- DosLoadModule
- DosQueryModuleName
- DosQueryProcType

DosQueryProcAddr - Example Code

This example loads the dynamic link module "DISPLAY.DDL", queries its address and type, and finally frees it.

```
 #define INCL_DOSMODULEMGR     /* Module Manager values */
 #define INCL_DOSERRORS        /* Error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

 PSZ      ModuleName     = "C:\\OS2\\DLL\\DISPLAY.DLL";  /* Name of module   */
 UCHAR    LoadError[256] = "";          /* Area for Load failure information */
 HMODULE  ModuleHandle   = NULLHANDLE;  /* Module handle                     */
 PFN      ModuleAddr     = 0;           /* Pointer to a system function      */
 ULONG    ModuleType     = 0;           /* Module type                       */
 APIRET   rc             = NO_ERROR;    /* Return code                       */

   rc = DosLoadModule(LoadError,               /* Failure information buffer */
                      sizeof(LoadError),       /* Size of buffer             */
                      ModuleName,              /* Module to load             */
                      &ModuleHandle);          /* Module handle returned     */
   if (rc != NO_ERROR) {
      printf("DosLoadModule error: return code = %u\n", rc);
      return 1;
   } else {
      printf("Module %s loaded.\n", ModuleName);
   } /* endif */

   rc = DosQueryProcAddr(ModuleHandle,         /* Handle to module           */
                         1L,                   /* No ProcName specified      */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleAddr);         /* Address returned           */
   if (rc != NO_ERROR) {
      printf("DosQueryProcAddr error: return code = %u\n", rc);
      return 1;
   } else printf("Address of module is 0x%x.\n", ModuleAddr);

   rc = DosQueryProcType(ModuleHandle,         /* Handle to module           */
                         1L,                   /* Indicate no ProcName given */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleType);         /* Type 0=16-bit   1=32-bit   */
   if (rc != NO_ERROR) {
      printf("DosQueryProcType error: return code = %u\n", rc);
      return 1;
   } else printf("This is a %s module.\n", ( ModuleType ? "32-bit" : "16-bit"));

   rc = DosFreeModule(ModuleHandle);
   if (rc != NO_ERROR) {
      printf("DosFreeModule error: return code = %u\n", rc);
      return 1;
   } else printf("Module %s freed.\n", ModuleName);

   return NO_ERROR;
}
```

DosQueryProcAddr - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryProcType

DosQueryProcType - Syntax

Returns the type of the specified procedure within a dynamic link module. The type returned indicates whether the specified procedure is a 16-bit or 32-bit callable procedure.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

HMODULE    hmod;         /*  The handle of the dynamic link module that contains the procedure. */
ULONG      ordinal;      /*  The ordinal number of the procedure whose type is desired. */
PSZ        pszName;      /*  The address of an ASCIIZ name string that contains the procedure name that is being
PULONG     pulproctype;  /*  The address of a ULONG in which the procedure type is returned. */
APIRET     ulrc;         /*  Return Code. */

ulrc = DosQueryProcType(hmod, ordinal, pszName,
         pulproctype);
```

DosQueryProcType Parameter - hmod

hmod (HMODULE) - input
The handle of the dynamic link module that contains the procedure.

DosQueryProcType Parameter - ordinal

ordinal (ULONG) - input
The ordinal number of the procedure whose type is desired.

If the ordinal number is nonzero, *pszName* is ignored.

DosQueryProcType Parameter - pszName

pszName (PSZ) - input
The address of an ASCIIZ name string that contains the procedure name that is being referenced.

Calls to DosQueryProcType for entries within the DOSCALLS module are supported for ordinal references only. References to the DOSCALLS module by name strings are not supported, and will return an error. Dynamic link ordinal numbers for DOSCALLS routines are resolved by linking with OS2386.LIB.

DosQueryProcType Parameter - pulproctype

pulproctype (PULONG) - output
The address of a ULONG in which the procedure type is returned.

The value returned in this field is one of the following:

0	PT_16BIT Procedure is 16-bit.
1	PT_32BIT Procedure is 32-bit.

DosQueryProcType Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryProcType returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
123	ERROR_INVALID_NAME
182	ERROR_INVALID_ORDINAL

For a full list of error codes, see Errors.

DosQueryProcType - Parameters

hmod (HMODULE) - input
The handle of the dynamic link module that contains the procedure.

ordinal (ULONG) - input
The ordinal number of the procedure whose type is desired.

If the ordinal number is nonzero, *pszName* is ignored.

pszName (PSZ) - input
The address of an ASCIIZ name string that contains the procedure name that is being referenced.

Calls to DosQueryProcType for entries within the DOSCALLS module are supported for ordinal references only. References to the DOSCALLS module by name strings are not supported, and will return an error. Dynamic link ordinal numbers for DOSCALLS routines are resolved by linking with OS2386.LIB.

pulproctype (PULONG) - output
The address of a ULONG in which the procedure type is returned.

The value returned in this field is one of the following:

0	PT_16BIT Procedure is 16-bit.
1	PT_32BIT Procedure is 32-bit.

ulrc (APIRET) - returns
Return Code.

DosQueryProcType returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
123	ERROR_INVALID_NAME
182	ERROR_INVALID_ORDINAL

For a full list of error codes, see Errors.

DosQueryProcType - Remarks

DosQueryProcType returns the type of the specified procedure within a dynamic link module.

The type returned indicates whether the specified procedure is a 16-bit or 32-bit callable procedure.

If return code ERROR_INVALID_HANDLE is received, issue DosLoadModule and then issue DosQueryProcType again.

DosQueryProcType - Related Functions

Related Functions

- DosFreeModule
- DosLoadModule
- DosQueryModuleName
- DosQueryProcAddr

DosQueryProcType - Example Code

This example loads the dynamic link module "DISPLAY.DDL," queries its address and type, and finally frees it.

```
 #define INCL_DOSMODULEMGR     /* Module Manager values */
 #define INCL_DOSERRORS        /* Error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

 PSZ      ModuleName     = "C:\\OS2\\DLL\\DISPLAY.DLL";  /* Name of module   */
 UCHAR    LoadError[256] = "";          /* Area for Load failure information */
 HMODULE  ModuleHandle   = NULLHANDLE;  /* Module handle                     */
 PFN      ModuleAddr     = 0;           /* Pointer to a system function      */
 ULONG    ModuleType     = 0;           /* Module type                       */
 APIRET   rc             = NO_ERROR;    /* Return code                       */

   rc = DosLoadModule(LoadError,              /* Failure information buffer */
                      sizeof(LoadError),      /* Size of buffer             */
                      ModuleName,             /* Module to load             */
                      &ModuleHandle);         /* Module handle returned     */
   if (rc != NO_ERROR) {
      printf("DosLoadModule error: return code = %u\n", rc);
      return 1;
   } else {
```

```
      printf("Module %s loaded.\n", ModuleName);
   } /* endif */

   rc = DosQueryProcAddr(ModuleHandle,         /* Handle to module           */
                         1L,                   /* No ProcName specified      */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleAddr);         /* Address returned           */
   if (rc != NO_ERROR) {
      printf("DosQueryProcAddr error: return code = %u\n", rc);
      return 1;
   } else printf("Address of module is 0x%x.\n", ModuleAddr);

   rc = DosQueryProcType(ModuleHandle,         /* Handle to module           */
                         1L,                   /* Indicate no ProcName given */
                         NULL,                 /* ProcName (not specified)   */
                         &ModuleType);         /* Type 0=16-bit   1=32-bit   */
   if (rc != NO_ERROR) {
      printf("DosQueryProcType error: return code = %u\n", rc);
      return 1;
   } else printf("This is a %s module.\n", ( ModuleType ? "32-bit" : "16-bit"));

   rc = DosFreeModule(ModuleHandle);
   if (rc != NO_ERROR) {
      printf("DosFreeModule error: return code = %u\n", rc);
      return 1;
   } else printf("Module %s freed.\n", ModuleName);

   return NO_ERROR;
}
```

DosQueryProcType - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryQueue

DosQueryQueue - Syntax

Queries the number of elements in a queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

HQUEUE    hq;          /*  The handle of the queue to be queried. */
PULONG    pcbEntries;  /*  A pointer to the number of elements in the queue. */
APIRET    ulrc;        /*  Return Code. */

ulrc = DosQueryQueue(hq, pcbEntries);
```

DosQueryQueue Parameter - hq

hq (HQUEUE) - input
The handle of the queue to be queried.

DosQueryQueue Parameter - pcbEntries

pcbEntries (PULONG) - output
A pointer to the number of elements in the queue.

DosQueryQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryQueue returns one of the following values:

0	NO_ERROR
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosQueryQueue - Parameters

hq (HQUEUE) - input
The handle of the queue to be queried.

pcbEntries (PULONG) - output
A pointer to the number of elements in the queue.

ulrc (APIRET) - returns
Return Code.

DosQueryQueue returns one of the following values:

0	NO_ERROR
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosQueryQueue - Remarks

DosQueryQueue returns the number of elements that are currently in a queue. This function can be used by the server process and its

threads, as well as by any client processes that have gained access to the queue by issuing DosOpenQueue.

If the server process closes the queue before this request is made, ERROR_QUE_INVALID_HANDLE is returned.

DosQueryQueue - Related Functions

Related Functions

- DosCloseQueue
- DosCreateQueue
- DosOpenQueue
- DosPeekQueue
- DosPurgeQueue
- DosReadQueue
- DosWriteQueue

DosQueryQueue - Example Code

This example shows how to create, write, querie, and close a queue.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSERRORS   /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

#define QUE_NAME "\\QUEUES\\PANDAWRITE\\LOCALQUEUE"

int main(VOID) {

   HQUEUE QueueHandle = NULLHANDLE;  /* Queue handle                    */
   CHAR   *DataBuffer = "";          /* Data to write to queue          */
   ULONG  ulNumElems  = 0L;          /* Number of elements on queue     */
   APIRET rc          = NO_ERROR;    /* Return code                     */

   rc = DosCreateQueue(&QueueHandle, /* Queue handle                    */
           QUE_LIFO |                /* Last In, First Out ordering     */
           QUE_CONVERT_ADDRESS,      /* Do 16-bit to 32-bit conversion */
           QUE_NAME);                /* Name of the queue               */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "Element 1 of 2";
   rc = DosWriteQueue (QueueHandle, 100L, strlen(DataBuffer),
                       (PVOID)DataBuffer, 0L);
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryQueue (QueueHandle, &ulNumElems);
   if (rc != NO_ERROR) {
      printf ("DosQueryQueue error: return code = %u\n", rc);
      return 1;
   } else { printf ("DosQueryQueue: %u elements\n", ulNumElems); }

   DataBuffer = "Element 2 of 2";
   rc = DosWriteQueue (QueueHandle, 200L, strlen(DataBuffer),
                       (PVOID)DataBuffer, 0L);
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryQueue (QueueHandle, &ulNumElems);
   if (rc != NO_ERROR) {
```

```
      printf ("DosQueryQueue error: return code = %u\n", rc);
      return 1;
   } else { printf ("DosQueryQueue: %u elements\n", ulNumElems); }
   rc = DosPurgeQueue (QueueHandle);
   if (rc != NO_ERROR) {
      printf ("DosPurgeQueue error: return code = %u\n", rc);
      return 1;        }

   rc = DosQueryQueue (QueueHandle, &ulNumElems);
   if (rc != NO_ERROR) {
      printf ("DosQueryQueue error: return code = %u\n", rc);
      return 1;
   } else { printf ("DosQueryQueue: %u elements\n", ulNumElems); }

   rc = DosCloseQueue(QueueHandle);      /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;       }

   return NO_ERROR;
}
```

DosQueryQueue - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryResourceSize

DosQueryResourceSize - Syntax

Returns the size of the specified resource object.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

HMODULE    hmod;     /*  The handle of the module that has the required resource. */
ULONG      idt;      /*  The type identifier of the resource. */
ULONG      idn;      /*  The name identifier of the resource. */
PULONG     pulsize;  /*  A pointer to a ULONG in which the size, in bytes, of the resource is returned. */
APIRET     ulrc;     /*  Return Code. */

ulrc = DosQueryResourceSize(hmod, idt, idn,
         pulsize);
```

DosQueryResourceSize Parameter - hmod

hmod (HMODULE) - input
The handle of the module that has the required resource.

A value of zero means to get the size from the current process. A value other than zero is a module handle that was returned by DosLoadModule.

DosQueryResourceSize Parameter - idt

idt (ULONG) - input
The type identifier of the resource.

Possible values are between 1 and 0xFFFE inclusive. Values from 1 to 255 are reserved for predefinition. Values from 256 and on can be type defined for the resource. The fist 21 values are predefined as follows:

1	RT_POINTER Mouse pointer shape
2	RT_BITMAP Bit map
3	RT_MENU Menu template
4	RT_DIALOG Dialog template
5	RT_STRING String tables
6	RT_FONTDIR Font directory
7	RT_FONT Font
8	RT_ACCELTABLE Accelerator tables
9	RT_RCDATA Binary data
10	RT_MESSAGE Error message tables
11	RT_DLGINCLUDE Dialog include file name
12	RT_VKEYTBL Key to vkey tables
13	RT_KEYTBL Key to UGL tables
14	RT_CHARTBL Glyph to character tables
15	RT_DISPLAYINFO Screen display information
16	RT_FKASHORT Function key area short form
17	RT_FKALONG Function key area long form
18	RT_HELPTABLE

Help table for Help manager

19 RT_HELPSUBTABLE
Help subtable for Help manager

20 RT_FDDIR
DBCS unique/font driver directory

21 RT_FD
DBCS unique/font driver

DosQueryResourceSize Parameter - idn

idn (ULONG) - input
The name identifier of the resource.

Possible values are between 1 and 0xFFFE inclusive.

DosQueryResourceSize Parameter - pulsize

pulsize (PULONG) - output
A pointer to a ULONG in which the size, in bytes, of the resource is returned.

DosQueryResourceSize Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryResourceSize returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosQueryResourceSize - Parameters

hmod (HMODULE) - input
The handle of the module that has the required resource.

A value of zero means to get the size from the current process. A value other than zero is a module handle that was returned by DosLoadModule.

idt (ULONG) - input
The type identifier of the resource.

Possible values are between 1 and 0xFFFE inclusive. Values from 1 to 255 are reserved for predefinition. Values from 256 and on can be type defined for the resource. The fist 21 values are predefined as follows:

1	RT_POINTER Mouse pointer shape
2	RT_BITMAP Bit map
3	RT_MENU Menu template
4	RT_DIALOG Dialog template
5	RT_STRING String tables
6	RT_FONTDIR Font directory
7	RT_FONT Font
8	RT_ACCELTABLE Accelerator tables
9	RT_RCDATA Binary data
10	RT_MESSAGE Error message tables
11	RT_DLGINCLUDE Dialog include file name
12	RT_VKEYTBL Key to vkey tables
13	RT_KEYTBL Key to UGL tables
14	RT_CHARTBL Glyph to character tables
15	RT_DISPLAYINFO Screen display information
16	RT_FKASHORT Function key area short form
17	RT_FKALONG Function key area long form
18	RT_HELPTABLE Help table for Help manager
19	RT_HELPSUBTABLE Help subtable for Help manager
20	RT_FDDIR DBCS unique/font driver directory
21	RT_FD DBCS unique/font driver

idn (ULONG) - input
The name identifier of the resource.

Possible values are between 1 and 0xFFFE inclusive.

pulsize (PULONG) - output
A pointer to a ULONG in which the size, in bytes, of the resource is returned.

ulrc (APIRET) - returns
Return Code.

DosQueryResourceSize returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosQueryResourceSize - Remarks

DosQueryResourceSize returns the size of the specified resource object.

Resource objects are read-only data objects that can be accessed dynamically at run time. The access key is two numbers. The first number is a type ID; the second, a name ID. These are similar to the file-extension and file-name portions of a file name.

Resource objects are placed into an executable file by the Resource Compiler (RC.EXE).

This function obtains the size of resources loaded from 16-bit executable files or dynamic link libraries (DLLs), since the size is not explicitly stored in most resources.

DosQueryResourceSize - Related Functions

Related Functions

- DosFreeResource
- DosGetResource
- DosLoadModule

DosQueryResourceSize - Example Code

This example loads the dynamic link module "DISPLAY.DLL," and queries the size of the font directory id=1 resource.

```
#define INCL_DOSRESOURCES     /* Resource types */
#define INCL_DOSMODULEMGR     /* Module Manager values */
#define INCL_DOSERRORS        /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR     LoadError[256] = "";            /* Area for Load failure information */
PSZ       ModuleName = "C:\\OS2\\DLL\\PMWP.DLL";  /* DLL with resources */
HMODULE   ModHandle  = NULLHANDLE;        /* Handle for module */
ULONG     Size       = 0;                 /* Resource size */
APIRET    rc         = NO_ERROR;          /* API return code */

 rc = DosLoadModule(LoadError,              /* Failure information buffer */
                    sizeof(LoadError),      /* Size of buffer             */
                    ModuleName,             /* Module to load             */
                    &ModHandle);            /* Module handle returned     */
 if (rc != NO_ERROR) {
    printf("DosLoadModule error: return code = %u\n", rc);
    return 1;
 }

 rc = DosQueryResourceSize(ModHandle,     /* Handle for DLL containing resources */
                           RT_POINTER,    /* Ask for  Pointer                    */
                           1L,            /*          with an ID of 1            */
                           &Size);        /* The resource size is returned.      */

 if (rc != NO_ERROR) {
```

```
    printf("DosGetResource error: return code = %u\n", rc);
    return 1;
 } else {
    printf("Resource is %u bytes in size.\n", Size);
 } /* endif */

return NO_ERROR;
}
```

DosQueryResourceSize - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQuerySysInfo

DosQuerySysInfo - Syntax

Returns values of static system variables.

```
#define INCL_DOSMISC
#include <os2.h>

ULONG     iStart;  /*  Ordinal of the first system variable to return. */
ULONG     iLast;   /*  Ordinal of the last system variable to return. */
PVOID     pBuf;    /*  Address of the data buffer where the system returns the variable values. */
ULONG     cbBuf;   /*  Length, in bytes, of the data buffer. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosQuerySysInfo(iStart, iLast, pBuf,
         cbBuf);
```

DosQuerySysInfo Parameter - iStart

iStart (ULONG) - input
Ordinal of the first system variable to return.

DosQuerySysInfo Parameter - iLast

iLast (ULONG) - input
Ordinal of the last system variable to return.

DosQuerySysInfo Parameter - pBuf

pBuf (PVOID) - output
Address of the data buffer where the system returns the variable values.

DosQuerySysInfo Parameter - cbBuf

cbBuf (ULONG) - input
Length, in bytes, of the data buffer.

DosQuerySysInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQuerySysInfo returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

For a full list of error codes, see Errors.

DosQuerySysInfo - Parameters

iStart (ULONG) - input
Ordinal of the first system variable to return.

iLast (ULONG) - input
Ordinal of the last system variable to return.

pBuf (PVOID) - output
Address of the data buffer where the system returns the variable values.

cbBuf (ULONG) - input
Length, in bytes, of the data buffer.

ulrc (APIRET) - returns
Return Code.

DosQuerySysInfo returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

For a full list of error codes, see Errors.

DosQuerySysInfo - Remarks

DosQuerySysInfo returns a single system variable or a range of system variables to a user-allocated buffer. To request a single system variable, set *iStart* equal to *iLast*. To request a range of system variables, set *iStart* less than *iLast*.

Each system variable is a ULONG value. The following list gives the ordinal index, name, and description of the system variables.

1 QSV_MAX_PATH_LENGTH
Maximum length, in bytes, of a path name.

2 QSV_MAX_TEXT_SESSIONS
Maximum number of text sessions.

3 QSV_MAX_PM_SESSIONS
Maximum number of PM sessions.

4 QSV_MAX_VDM_SESSIONS
Maximum number of DOS sessions.

5 QSV_BOOT_DRIVE
Drive from which the system was started (1 means drive A, 2 means drive B, and so on).

6 QSV_DYN_PRI_VARIATION
Dynamic priority variation flag (0 means absolute priority, 1 means dynamic priority).

7 QSV_MAX_WAIT
Maximum wait in seconds.

8 QSV_MIN_SLICE
Minimum time slice in milliseconds.

9 QSV_MAX_SLICE
Maximum time slice in milliseconds.

10 QSV_PAGE_SIZE
Memory page size in bytes. This value is 4096 for the 80386 processor.

11 QSV_VERSION_MAJOR
Major version number (see note below).

12 QSV_VERSION_MINOR
Minor version number (see note below).

13 QSV_VERSION_REVISION
Revision number (see note below).

14 QSV_MS_COUNT
Value of a 32-bit, free-running millisecond counter. This value is zero when the system is started.

15 QSV_TIME_LOW
Low-order 32 bits of the time in seconds since January 1, 1970 at 0:00:00.

16 QSV_TIME_HIGH
High-order 32 bits of the time in seconds since January 1, 1970 at 0:00:00.

17 QSV_TOTPHYSMEM
Total number of bytes of physical memory in the system.

18 QSV_TOTRESMEM
Total number of bytes of resident memory in the system.

19 QSV_TOTAVAILMEM
Maximum number of bytes of memory that can be allocated by all processes in the system. This number is advisory and is not

guaranteed, since system conditions change constantly.

20 QSV_MAXPRMEM
Maximum number of bytes of memory that this process can allocate in its private arena. This number is advisory and is not guaranteed, since system conditions change constantly.

21 QSV_MAXSHMEM
Maximum number of bytes of memory that a process can allocate in the shared arena. This number is advisory and is not guaranteed, since system conditions change constantly.

22 QSV_TIMER_INTERVAL
Timer interval in tenths of a millisecond.

23 QSV_MAX_COMP_LENGTH
Maximum length, in bytes, of one component in a path name.

24 QSV_FOREGROUND_FS_SESSION
Session ID of the current foreground full-screen session. Note that this only applies to full-screen sessions. The Presentation Manager session (which displays Vio-windowed, PM, and windowed DOS Sessions) is full-screen session ID 1.

25 QSV_FOREGROUND_PROCESS
Process ID of the current foreground process.

Note: Major, minor and revision numbers for versions of OS/2 operating system are described below:

	Major	Minor	Revision
OS/2 2.0	20	00	0
OS/2 2.1	20	10	0
OS/2 2.11	20	11	0
OS/2 3.0	20	30	0
OS/2 4.0	20	40	0

An application can specify file objects managed by an installable file system that supports long file names. Because some installable file systems support longer names than others, the application should issue DosQuerySysInfo upon initialization.

DosQuerySysInfo returns the maximum path length (QSV_MAX_PATH_LENGTH) supported by the installed file system. The path length includes the drive specifier (d:), the leading backslash (\), and the trailing null character. The value returned by DosQuerySysInfo can be used to allocate buffers for path names returned by other functions, for example, DosFindFirst and DosFindNext.

DosQuerySysInfo - Related Functions

Related Functions

- DosCreateDir
- DosFindFirst
- DosFindNext
- DosOpen
- DosQueryCurrentDir
- DosQueryFSInfo
- DosQueryPathInfo
- DosSearchPath
- DosSetCurrentDir
- DosSetPathInfo
- DosSetFSInfo

DosQuerySysInfo - Example Code

This example queries and displays the maximum length for a path name and the total amount of physical memory in bytes.

```
#define INCL_DOSMISC       /* DOS Miscellaneous values */
#define INCL_DOSERRORS     /* DOS Error values         */
#include <os2.h>
#include <stdio.h>

int main(VOID)  {

  ULONG   aulSysInfo[QSV_MAX] = {0};       /* System Information Data Buffer */
  APIRET  rc                  = NO_ERROR;  /* Return code                    */

  rc = DosQuerySysInfo(1L,                 /* Request all available system   */
                       QSV_MAX,            /* information                    */
                       (PVOID)aulSysInfo,
                       sizeof(ULONG)*QSV_MAX);

  if (rc != NO_ERROR) {
     printf("DosQuerySysInfo error: return code = %u\n", rc);
     return 1;
  } else {
     printf("Maximum length for a path name is %u characters.\n",
             aulSysInfo[QSV_MAX_PATH_LENGTH-1]);  /* Max length of path name */

     printf("Total physical memory is %u bytes.\n",
             aulSysInfo[QSV_TOTPHYSMEM-1]);       /* Total physical memory   */
  } /* endif */

   return NO_ERROR;
}
```

DosQuerySysInfo - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryThreadContext

DosQueryThreadContext - Syntax

Query context of a suspended thread.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

TID              tid;    /*  Thread identity. */
ULONG            level;  /*  Level of information desired. */
PCONTEXTRECORD   pcxt;   /*  Thread context record. */
APIRET           ulrc;   /*  Return Code. */

ulrc = DosQueryThreadContext(tid, level, pcxt);
```

DosQueryThreadContext Parameter - tid

tid (TID) - input
Thread identity.

DosQueryThreadContext Parameter - level

level (ULONG) - input
Level of information desired:

CONTEXT_CONTROL (0x00000001)
Control registers: SS:ESP, CS:EIP, EFLAGS and EBP

CONTEXT_INTEGER (0x00000002)
EAX, EBX, ECX, EDX, ESI and EDI

CONTEXT_SEGMENTS (0x00000004)
Segment registers: DS, ES, FS, and GS

CONTEXT_FLOATING_POINT (0x00000008)
Numeric coprocessor state

CONTEXT_FULL
All of the above

DosQueryThreadContext Parameter - pcxt

pcxt (PCONTEXTRECORD) - in/out
Thread context record.

DosQueryThreadContext Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryThreadContext returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
90	ERROR_NOT_FROZEN
115	ERROR_PROTECTION_VIOLATION
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosQueryThreadContext - Parameters

tid (TID) - input
Thread identity.

level (ULONG) - input
Level of information desired:

CONTEXT_CONTROL (0x00000001) Control registers: SS:ESP, CS:EIP, EFLAGS and EBP

CONTEXT_INTEGER (0x00000002) EAX, EBX, ECX, EDX, ESI and EDI

CONTEXT_SEGMENTS (0x00000004) Segment registers: DS, ES, FS, and GS

CONTEXT_FLOATING_POINT (0x00000008) Numeric coprocessor state

CONTEXT_FULL All of the above

pcxt (PCONTEXTRECORD) - in/out
Thread context record.

ulrc (APIRET) - returns
Return Code.

DosQueryThreadContext returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
90	ERROR_NOT_FROZEN
115	ERROR_PROTECTION_VIOLATION
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosQueryThreadContext - Remarks

DosQueryThreadContext returns the context record of a suspended thread. A thread may be suspended by using DosSuspendThread or DosEnterCritSec. If DosSuspendThread is used, the caller must allow some time for OS/2 to suspend the thread before querying its context.

Note: Values from the thread context should be used only when the state of the target thread is known.

DosQueryThreadContext - Related Functions

Related Functions

- DosCreateThread
- DosEnterCritSec
- DosResumeThread
- DosSuspendThread

DosQueryThreadContext - Example Code

This example uses DosQueryThreadContext to query the context of a suspended thread.

```
#define INCL_DOSEXCEPTIONS
#define INCL_DOSPROCESS
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

VOID _System CntThreadProc( ULONG LoopMax );             /* Count Thread */
VOID PrintContextRecord( CONTEXTRECORD *pThread );

int main(VOID)
{
    TID        tidCntThread = 0;   /* ID returned for newly created thread */
    PFNTHREAD  pfnCntThread = &CntThreadProc; /* Address of Thread program */
    ULONG      ulLoopMax    = 500000;     /* Parameter to thread routine   */
    APIRET     rc           = NO_ERROR; /* Return code                  */

                      /* Thread Context Record */
    CONTEXTRECORD  Thread_cxt= {0,{0,0,0,0,0,0,0},
                               {{0,0,0}, {0,0,0}, {0,0,0}, {0,0,0},
                                {0,0,0}, {0,0,0}, {0,0,0}, {0,0,0}},
                                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    rc = DosCreateThread(&tidCntThread, /* Thread ID (return by function) */
                         pfnCntThread,   /* Address of thread program      */
                         ulLoopMax,      /* Parameter passed to ThreadProc */
                         CREATE_READY | /* Thread is ready when created */
                         STACK_SPARSE, /* Do not pre_commit stack pages */
                         8192L);         /* Stack size, rounded to page bdy*/
    if (rc != NO_ERROR) {
        printf("DosCreateThread error: return code = %u\n",rc);
        return 1;
    }

    rc = DosSuspendThread (tidCntThread);
    if (rc != NO_ERROR)
        printf("DosSuspendThread error: return code = %u\n",rc);
    else
    {           /* Wait for thread to suspend */

        printf("Waiting 5 seconds for thread to suspend...\n");
        rc = DosSleep(5000);

        rc = DosQueryThreadContext (tidCntThread,
                                    CONTEXT_FULL,
                                    &Thread_cxt);
        if (rc != NO_ERROR)
           printf("DosQueryThreadContext error: return code = %u\n",rc);
        else
           PrintContextRecord( &Thread_cxt );


        rc = DosResumeThread (tidCntThread);
        if (rc != NO_ERROR) {
            printf("DosResumeThread error: return code = %u\n",rc);
            return 1;
        }
    }

    printf("Waiting for thread to complete...\n");

    rc = DosWaitThread ( &tidCntThread, DCWW_WAIT );
    if (rc != NO_ERROR) {
        printf("DosWaitThread error: return code = %u\n",rc);
        return 1;
    }

    printf("Thread has completed!\n");
    return NO_ERROR;
}

VOID _System CntThreadProc( ULONG LoopMax ) /* Count Thread */
{
    ULONG  ulCount = 0,
           i       = 0,
           j       = 1,
           k       = 1,
           l       = 0;
    double w       = 0.0,
           y       = 0.0,
           z       = 0.0;

    for (l=0; l < 20; l++)
```

```
    {
      for (i=0; i < LoopMax; i++)
      {
        w = 1.00 / 2.00;
        y = 2.00 * 2.00;
        z = 3.00 / 6.00;
        j = j + i;
        k = k * i;
      }
    }

  return;
}

VOID PrintContextRecord( CONTEXTRECORD *pThread )
{
       int i = 0;

       printf("\nRegisters for CONTEXT_CONTROL level:\n");
       printf("  EBP= 0x%08x\t",pThread->ctx_RegEbp);
       printf("EIP= 0x%08x\t",pThread->ctx_RegEip);
       printf("CS= 0x%08x\n",pThread->ctx_SegCs);
       printf("  EFLAGS = 0x%08x\t",pThread->ctx_EFlags);
       printf("ESP= 0x%08x\t",pThread->ctx_RegEsp);
       printf("SS= 0x%08x\n",pThread->ctx_SegSs);

       printf("\nRegisters for CONTEXT_INTEGER level:\n");
       printf("  EAX= 0x%08x\t",pThread->ctx_RegEax);
       printf("EBX= 0x%08x\t",pThread->ctx_RegEbx);
       printf("ECX= 0x%08x\n",pThread->ctx_RegEcx);
       printf("  EDX= 0x%08x\t",pThread->ctx_RegEdx);
       printf("ESI= 0x%08x\t",pThread->ctx_RegEsi);
       printf("EDI= 0x%08x\n",pThread->ctx_RegEdi);

       printf("\nRegisters for CONTEXT_SEGMENTS level:\n");
       printf("  DS= 0x%08x\t",pThread->ctx_SegDs);
       printf("ES= 0x%08x\t",pThread->ctx_SegEs);
       printf("FS= 0x%08x\t",pThread->ctx_SegFs);
       printf("GS= 0x%08x\n",pThread->ctx_SegGs);

       printf("\nRegisters for CONTEXT_FLOATING_POINT level:\n");
       printf("  - Coprocessor stack register (exponent,hisig:losig):\n");
       for (i=0;i<8;i++)
       {
          printf("    ctx_stack[%d]=0x%08x ,0x%08x :0x%08x \n",i,
                  pThread->ctx_stack[i].signexp,
                  pThread->ctx_stack[i].hisig,
                  pThread->ctx_stack[i].losig);
       }
       printf("  - Other element\n");
       for (i=0;i<7;i++)
       {
          printf("    ctx_env[%d]=0x%08x \n",i ,pThread->ctx_env[i]);
       }

       return;
}
```

DosQueryThreadContext - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosQueryVerify

DosQueryVerify - Syntax

Determines if write verification is enabled.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PBOOL32    pBool;  /*  Address of the verify mode for the process. */
APIRET     ulrc;   /*  Return Code. */

ulrc = DosQueryVerify(pBool);
```

DosQueryVerify Parameter - pBool

pBool (PBOOL32) - output
Address of the verify mode for the process.

Possible values are shown in the list below:

0	Verify mode is not active.
1	Verify mode is active.

DosQueryVerify Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosQueryVerify returns one of the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosQueryVerify - Parameters

pBool (PBOOL32) - output
Address of the verify mode for the process.

Possible values are shown in the list below:

0	Verify mode is not active.
1	Verify mode is active.

ulrc (APIRET) - returns

Return Code.

DosQueryVerify returns one of the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosQueryVerify - Remarks

When the verify mode is active, the operating system verifies that data written to the disk is recorded correctly, even though disk recording errors are rare.

DosQueryVerify - Related Functions

Related Functions

- DosSetVerify

DosQueryVerify - Example Code

This example enables disk write verification, and then creates and writes to a file named "VERIFY.DAT". It restores the original setting when it finishes.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   UCHAR  uchFileName[20]  = "VERIFY.DAT",     /* Name of file to use      */
          uchFileData[10] = "SampleData";      /* Stuff to put in file     */
   HFILE  hfFileHandle   = 0L;                 /* Handle for user file     */
   BOOL32 fUserVerify    = 0;                  /* User Verify flag setting */
   ULONG  ulAction       = 0,                  /* Action done by DosOpen   */
          ulWritten      = 0;                  /* Number of bytes written  */
   APIRET rc             = NO_ERROR;           /* Return code              */

   rc = DosQueryVerify(&fUserVerify);     /* Get current setting of VERIFY */
   if (rc != NO_ERROR) {
     printf("DosQueryVerify error: return code = %u\n", rc);
     return 1;
   } else {
     printf ("Original setting of Verify=%s\n",(fUserVerify) ? "On" : "Off");
   } /* endif */

   rc = DosSetVerify(1);                  /* Set VERIFY=ON                */
   if (rc != NO_ERROR) {
     printf("DosSetVerify error: return code = %u\n", rc);
     return 1;  }

   /* Open the file VERIFY.DAT for read/write.  Create it if necessary */

   rc = DosOpen(uchFileName, &hfFileHandle, &ulAction,
                10L, FILE_NORMAL,
                OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_SHARE_DENYNONE  | OPEN_ACCESS_READWRITE, 0L);

   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
```

```
      return 1; }

         /* Write critical data to the file */

   rc = DosWrite (hfFileHandle, (PVOID) uchFileData,
                  sizeof(uchFileData), &ulWritten);
   if (rc == NO_ERROR) {
      printf ("%u bytes written to file %s with Verify=On\n",
               ulWritten,uchFileName);
   }
   rc = DosSetVerify(fUserVerify);         /* Restore user's verify value */
   if (rc != NO_ERROR) {
     printf("DosSetVerify error: return code = %u\n", rc);
     return 1;
   }

   rc = DosClose(hfFileHandle);            /* Close the file              */

   return NO_ERROR;
}
```

DosQueryVerify - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosRaiseException

DosRaiseException - Syntax

Raises an exception for the current thread.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PEXCEPTIONREPORTRECORD    pexcept;  /*  A pointer to an exception report record that contains exception-specific
APIRET                    ulrc;     /*  Return Code. */

ulrc = DosRaiseException(pexcept);
```

DosRaiseException Parameter - pexcept

pexcept (PEXCEPTIONREPORTRECORD) - input
A pointer to an exception report record that contains exception-specific information needed for the exception to be raised.

The pointer to the exception report record, as well as certain handler flags in the structure, are supplied by the system.

DosRaiseException Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosRaiseException returns the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosRaiseException - Parameters

pexcept (PEXCEPTIONREPORTRECORD) - input
A pointer to an exception report record that contains exception-specific information needed for the exception to be raised.

The pointer to the exception report record, as well as certain handler flags in the structure, are supplied by the system.

ulrc (APIRET) - returns
Return Code.

DosRaiseException returns the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosRaiseException - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosRaiseException enables a thread to raise a synchronous exception that has been deferred from a must-complete section. DosRaiseException can also be used to simulate an asynchronous or synchronous exception.

For a detailed list of the system exceptions, see System Exceptions.

DosRaiseException - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosExitMustComplete
- DosSendSignalException

- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler
- DosUnwindException

DosRaiseException - Example Code

This example shows how to raise an exception for the current thread.

```
#define INCL_DOSPROCESS        /* DOS process values (for DosSleep) */
#define INCL_DOSEXCEPTIONS     /* DOS exception values */
#define INCL_ERRORS            /* DOS error values     */
#include <os2.h>
#include <stdio.h>

ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv );
int main (VOID)
{
  EXCEPTIONREGISTRATIONRECORD RegRec  = {0};     /* Exception Registration Record */
  APIRET      rc      = NO_ERROR;   /* Return code                   */

  /* Add MyTermHandler to this thread's chain of exception handlers */

  RegRec.ExceptionHandler = (ERR)MyTermHandler;
  rc = DosSetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosSetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Terminate this program using Ctrl-C or Ctrl-Break.\n");

  rc = DosSleep(60000L);   /* Give user plenty of time to comply... */

  rc = DosUnsetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosUnsetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }
  return NO_ERROR;
}
/*********************************************************************/
ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv )
{
  APIRET  rc = NO_ERROR;

  printf("*** MyTermHandler entered: ExceptionNum = %x\n",p1->ExceptionNum);

  rc = DosUnsetExceptionHandler( p2 );  /* Stop recursive entry to handler */
  rc = DosRaiseException( p1 );         /* Exception Report Record */
  if (rc != NO_ERROR) {
     printf("DosRaiseException error:  return code = %u\n", rc);
  }

  return XCPT_CONTINUE_SEARCH;
}
```

DosRaiseException - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosRead

DosRead - Syntax

Reads the specified number of bytes from a file, pipe, or device to a buffer location.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;      /*  File handle obtained from DosOpen. */
PVOID     pBuffer;    /*  Address of the buffer to receive the bytes read. */
ULONG     cbRead;     /*  The length, in bytes, of pBuffer. */
PULONG    pcbActual;  /*  Address of the variable to receive the number of bytes actually read. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosRead(hFile, pBuffer, cbRead, pcbActual);
```

DosRead Parameter - hFile

hFile (HFILE) - input
File handle obtained from DosOpen.

DosRead Parameter - pBuffer

pBuffer (PVOID) - output
Address of the buffer to receive the bytes read.

DosRead Parameter - cbRead

cbRead (ULONG) - input
The length, in bytes, of *pBuffer* .

This is the number of bytes to be read.

DosRead Parameter - pcbActual

pcbActual (PULONG) - output
Address of the variable to receive the number of bytes actually read.

DosRead Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosRead returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
232	ERROR_NO_DATA
234	ERROR_MORE_DATA

For a full list of error codes, see Errors.

DosRead - Parameters

hFile (HFILE) - input
File handle obtained from DosOpen.

pBuffer (PVOID) - output
Address of the buffer to receive the bytes read.

cbRead (ULONG) - input
The length, in bytes, of *pBuffer* .

This is the number of bytes to be read.

pcbActual (PULONG) - output
Address of the variable to receive the number of bytes actually read.

ulrc (APIRET) - returns
Return Code.

DosRead returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
232	ERROR_NO_DATA
234	ERROR_MORE_DATA

For a full list of error codes, see Errors.

DosRead - Remarks

Note: When writing message pipes the application is limited to 64K messages. As well, these messages cannot span 64k boundaries due to the current design of the thunk layer in read or write routines. If the message is not written in an aligned manner, the subsequent read will not be able to handle the messages properly. If a 64k or less message is written to a pipe from an aligned buffer, the read will handle this properly.

The requested number of bytes might not be read. If the value returned in *pcbActual* is zero, the process tried to read from the end of the file.

A value of zero for *cbRead* is not considered an error. In such a case, the system treats the request as a null operation.

The file pointer is moved to the desired position by reading data, writing data, or issuing DosSetFilePtr.

If you issue DosOpen with the Direct Open flag set to 1 in the *fsOpenFlags* parameter, you have direct access to an entire disk or diskette volume, independent of the file system. You must lock the logical volume before accessing it, and you must unlock the logical volume when you are finished accessing it. Issue DosDevIOCtl for Category 8, DSK_LOCKDRIVE to lock the logical volume, and for Category 8, DSK_UNLOCKDRIVE to unlock the logical volume. While the logical volume is locked, no other process can access it.

Named Pipe Considerations

A named pipe is read as one of the following:

- A byte pipe in byte-read mode
- A message pipe in message-read mode
- A message pipe in byte-read mode.

A byte pipe must be in byte-read mode to be read; an error is returned if it is in message-read mode. All currently available data, up to the size requested, is returned.

A message pipe can be read in either message-read mode or byte-read mode. When the message pipe is in message-read mode, a read operation that is larger than the next available message returns only that message. *pcbActual* is set to indicate the size of the message returned.

A read operation that is smaller than the next available message returns with the number of bytes requested and an ERROR_MORE_DATA return code. When the reading of a message is resumed after ERROR_MORE_DATA is returned, a read operation always blocks until the next piece (or the rest) of the message can be transferred. DosPeekNPipe can be issued to determine how many bytes are left in the message.

A message pipe in byte-read mode is read as if it were a byte stream, and DosRead skips over message headers. This is like reading a byte pipe in byte-read mode.

When blocking mode is set for a named pipe, a read operation blocks until data is available. In this case, the read operation never returns with *pcbActual* equal to zero, except when the pipe is closed. When the mode is set to message-read, messages are always read in their entirety, except when the message is bigger than the size of the read operation.

pcbActual can equal zero in nonblocking mode, but only when no data is available at the time of the read operation.

If you attempt a read of a named pipe when the other end of the pipe is closed while the read is pending, DosRead returns:

- pcbActual equal to 0
- ulrc equal to either 0 or ERROR_NO_DATA

Unnamed Pipe Considerations

Threads writing to and reading from the same unnamed pipe are not synchronized. It is possible for the pipe reader to obtain partial contents of a write request if the pipe becomes full during the write request.

DosRead - Related Functions

Related Functions

- DosOpen

- DosSetFilePtr
- DosWrite

DosRead - Example Code

This example opens or creates and opens a file named "DOSTEST.DAT", writes to it, reads from it, and finally closes the file.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(void) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  ulBytesRead    = 0;      /* Number of bytes read by DosRead */
   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   ULONG  ulLocal        = 0;      /* File pointer position after DosSetFilePtr */
   UCHAR  uchFileName[20]  = "dostest.dat",     /* Name of file */
          uchFileData[100] = " ";               /* Data to write to file */
   APIRET rc              = NO_ERROR;           /* Return code */

   /* Open the file dostest.dat. Use an existing file or create a new */
   /* one if it doesn't exist.                                        */
   rc = DosOpen(uchFileName,                    /* File path name */
                &hfFileHandle,                  /* File handle */
                &ulAction,                      /* Action taken */
                100L,                           /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
                OPEN_ACTION_CREATE_IF_NEW |
                OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type */
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file */
                0L);                            /* No extended attribute */

   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosOpen: Action taken = %ld\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n1...\n2...\n3\n");

   rc = DosWrite (hfFileHandle,                 /* File handle */
                  (PVOID) uchFileData,          /* String to be written */
                  sizeof (uchFileData),         /* Size of string to be written */
                  &ulWrote);                    /* Bytes actually written */
   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosSetFilePtr (hfFileHandle,            /* File Handle */
                       0L,                      /* Offset */
                       FILE_BEGIN,              /* Move from BOF */
                       &ulLocal);               /* New location address */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosRead (hfFileHandle,                  /* File Handle */
                 uchFileData,                   /* String to be read */
                 100L,                          /* Length of string to be read */
                 &ulBytesRead);                 /* Bytes actually read */

   if (rc != NO_ERROR) {
```

```
      printf("DosRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosClose(hfFileHandle);               /* Close the file */

   if (rc != NO_ERROR) {
      printf("DosClose error: return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosRead - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosReadQueue

DosReadQueue - Syntax

Reads an element from a queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

HQUEUE          hq;         /*  The handle of the queue from which an element is to be removed. */
PREQUESTDATA    pRequest;   /*  A pointer to a REQUESTDATA that returns a PID and an event code. */
PULONG          pcbData;    /*  A pointer to the length, in bytes, of the data that is being removed. */
PPVOID          ppbuf;      /*  A pointer to the element that is being removed from the queue. */
ULONG           element;    /*  An indicator that specifies whether to remove the first element in the queue or
BOOL32          wait;       /*  The action to be performed when no entries are found in the queue. */
PBYTE           ppriority;  /*  The address of the element's priority value. */
HEV             hsem;       /*  The handle of an event semaphore that is to be posted when data is added to the
APIRET          ulrc;       /*  Return Code. */

ulrc = DosReadQueue(hq, pRequest, pcbData,
         ppbuf, element, wait, ppriority,
         hsem);
```

DosReadQueue Parameter - hq

hq (HQUEUE) - input
The handle of the queue from which an element is to be removed.

DosReadQueue Parameter - pRequest

pRequest (PREQUESTDATA) - output
A pointer to a REQUESTDATA that returns a PID and an event code.

The data in the *ulData* field of the REQUESTDATA structure is the same as the data that was furnished in the *request* parameter of DosWriteQueue for the corresponding queue element.

DosReadQueue Parameter - pcbData

pcbData (PULONG) - output
A pointer to the length, in bytes, of the data that is being removed.

DosReadQueue Parameter - ppbuf

ppbuf (PPVOID) - output
A pointer to the element that is being removed from the queue.

(This field may or may not be the same as the value of *pbData* that was specified with DosWriteQueue when the element was added to the queue. If QUE_CONVERT_ADDRESS was specified when the queue was created, the addresses of any elements that are written to the queue by the 16-bit DosWriteQueue function are converted to 32-bit addresses.)

DosReadQueue Parameter - element

element (ULONG) - input
An indicator that specifies whether to remove the first element in the queue or the queue element that was previously examined by DosPeekQueue.

Possible values are shown in the following list:

0	This field is set to 0 by the application to indicate "remove the first element in the queue," according to the order that was specified when the queue was created (FIFO, LIFO, or priority).
Any other value	The field is set to the value that was returned by a previous DosPeekQueue operation to indicate "remove the element that was examined by DosPeekQueue."

DosReadQueue Parameter - wait

wait (BOOL32) - input
The action to be performed when no entries are found in the queue.

Possible values are shown in the following list:

0	DCWW_WAIT The requesting thread waits for an element to be added to the queue.
1	DCWW_NOWAIT The requesting thread does not wait, and DosReadQueue returns with ERROR_QUE_EMPTY.

DosReadQueue Parameter - ppriority

ppriority (PBYTE) - output
The address of the element's priority value.

This is the value that was specified for *ppriority* by DosWriteQueue when it added the element to the queue. *ppriority* is a numerical value in the range of 0 to 15, with 15 being the highest priority.

DosReadQueue Parameter - hsem

hsem (HEV) - input
The handle of an event semaphore that is to be posted when data is added to the queue and *wait* is set to 1.

This parameter is ignored if *wait* is set to 0.

The event semaphore may be shared or private, depending on whether the queue is shared across processes.

Note: The first time an event-semaphore handle is supplied in a DosReadQueue or DosPeekQueue request for which *wait* is set to 1, the handle is saved by the system. The same handle must be supplied in all subsequent DosReadQueue and DosPeekQueue requests that are issued for that queue.

DosReadQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosReadQueue returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
330	ERROR_QUE_PROC_NOT_OWNED
333	ERROR_QUE_ELEMENT_NOT_EXIST
337	ERROR_QUE_INVALID_HANDLE
342	ERROR_QUE_EMPTY
433	ERROR_QUE_INVALID_WAIT

For a full list of error codes, see Errors.

DosReadQueue - Parameters

hq (HQUEUE) - input
The handle of the queue from which an element is to be removed.

pRequest (PREQUESTDATA) - output
A pointer to a REQUESTDATA that returns a PID and an event code.

The data in the *ulData* field of the REQUESTDATA structure is the same as the data that was furnished in the *request* parameter of DosWriteQueue for the corresponding queue element.

pcbData (PULONG) - output
A pointer to the length, in bytes, of the data that is being removed.

ppbuf (PPVOID) - output
A pointer to the element that is being removed from the queue.

(This field may or may not be the same as the value of *pbData* that was specified with DosWriteQueue when the element was added to the queue. If QUE_CONVERT_ADDRESS was specified when the queue was created, the addresses of any elements that are written to the queue by the 16-bit DosWriteQueue function are converted to 32-bit addresses.)

element (ULONG) - input
An indicator that specifies whether to remove the first element in the queue or the queue element that was previously examined by DosPeekQueue.

Possible values are shown in the following list:

0	This field is set to 0 by the application to indicate "remove the first element in the queue," according to the order that was specified when the queue was created (FIFO, LIFO, or priority).
Any other value	The field is set to the value that was returned by a previous DosPeekQueue operation to indicate "remove the element that was examined by DosPeekQueue."

wait (BOOL32) - input
The action to be performed when no entries are found in the queue.

Possible values are shown in the following list:

0	DCWW_WAIT The requesting thread waits for an element to be added to the queue.
1	DCWW_NOWAIT The requesting thread does not wait, and DosReadQueue returns with ERROR_QUE_EMPTY.

ppriority (PBYTE) - output
The address of the element's priority value.

This is the value that was specified for *ppriority* by DosWriteQueue when it added the element to the queue. *ppriority* is a numerical value in the range of 0 to 15, with 15 being the highest priority.

hsem (HEV) - input
The handle of an event semaphore that is to be posted when data is added to the queue and *wait* is set to 1.

This parameter is ignored if *wait* is set to 0.

The event semaphore may be shared or private, depending on whether the queue is shared across processes.

Note: The first time an event-semaphore handle is supplied in a DosReadQueue or DosPeekQueue request for which *wait* is set to 1, the handle is saved by the system. The same handle must be supplied in all subsequent DosReadQueue and DosPeekQueue requests that are issued for that queue.

ulrc (APIRET) - returns
Return Code.

DosReadQueue returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
330	ERROR_QUE_PROC_NOT_OWNED
333	ERROR_QUE_ELEMENT_NOT_EXIST

337	ERROR_QUE_INVALID_HANDLE
342	ERROR_QUE_EMPTY
433	ERROR_QUE_INVALID_WAIT

For a full list of error codes, see Errors.

DosReadQueue - Remarks

DosReadQueue reads (removes) an element from a queue. This function can be issued only by the server process and its threads.

If the *wait* parameter is set to 1, an event semaphore must be provided so that the calling thread can determine when an element has been placed into the queue. The semaphore is created by calling DosCreateEventSem, and its handle is supplied in the *hsem* parameter of DosReadQueue.

The first time an event-semaphore handle is supplied in a DosReadQueue or DosPeekQueue request for which *wait* has been set to 1, the handle is saved by the system. The same handle must be supplied in all subsequent DosReadQueue and DosPeekQueue requests that are issued for the same queue; if a different handle is supplied, ERROR_INVALID_PARAMETER is returned.

When a client process adds an element to the queue, the system automatically opens and posts the semaphore. The server can either issue DosQueryEventSem periodically to determine whether the semaphore has been posted, or it can issue DosWaitEventSem. DosWaitEventSem causes the calling thread to block until the semaphore is posted.

After the event semaphore has been posted, the calling thread must issue DosReadQueue again to remove the newly added queue element.

DosReadQueue - Related Functions

Related Functions

- DosCloseQueue
- DosCreateQueue
- DosOpenQueue
- DosPeekQueue
- DosPurgeQueue
- DosQueryQueue
- DosWriteQueue

DosReadQueue - Example Code

This example creates and opens a queue named "SPECIAL.QUE", writes to it, reads from it, and then closes it.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSPROCESS  /* DOS thread and process values */
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

   PSZ         szQueueName  = "\\QUEUES\\SPECIAL.QUE";
   HQUEUE      hqSpecialQue = NULLHANDLE; /* Queue handle                    */
   PSZ         DataBuffer   = "";         /* Data buffer for queue data      */
   REQUESTDATA Request      = {0};        /* Reques */
   PID         pidOwner     = 0;
   ULONG       ulDataLen    = 0;          /* Length of data returned         */
   BYTE        ElemPrty     = 0;          /* Priority of element (returned)  */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateQueue(&hqSpecialQue,    /* Queue handle                    */
            QUE_FIFO |                   /* First-In First-Out order        */
```

```
            QUE_CONVERT_ADDRESS,        /* Convert 16-bit addresses to 32  */
            szQueueName);               /* Name of the queue to create     */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosOpenQueue (&pidOwner,          /* PID of queue owner              */
                      &hqSpecialQue,      /* Handle for created queue        */
                      szQueueName);       /* Name of the queue to open       */
   if (rc!= NO_ERROR) {
      printf ("DosOpenQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "To be, or not to be.  That is the question...";
   rc = DosWriteQueue (hqSpecialQue,        /* Queue to write to           */
                       12345L,              /* Request data                */
                       strlen(DataBuffer),  /* Length of data to write     */
                       DataBuffer,          /* Pointer to data             */
                       0L);              /* Priority (not applicable here) */
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "";                  /* Clear the DataBuffer */
   Request.pid = pidOwner;           /* process ID for the DosReadQueue */

   rc = DosReadQueue (hqSpecialQue,          /* Queue to read from          */
                      &Request,              /* Request data from write     */
                      &ulDataLen,            /* Length of data returned     */
                      (PVOID) &DataBuffer,   /* The data                    */
                      0L,                    /* Remove first element        */
                      DCWW_WAIT,             /* Wait for available data     */
                      &ElemPrty,             /* Priority of data (returned) */
                      0L);             /* Semaphore to use when not waiting */
   if (rc!= NO_ERROR) {
      printf ("DosReadQueue error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosReadQueue returns: '%s'\n", DataBuffer);
      printf ("      (Request data = %u)\n", Request.ulData);
   }

   rc = DosCloseQueue(hqSpecialQue);         /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosReadQueue - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosReleaseMutexSem

DosReleaseMutexSem - Syntax

Relinquishes ownership of a mutex semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMTX      hmtx;  /*  The handle of the mutex semaphore to release. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosReleaseMutexSem(hmtx);
```

DosReleaseMutexSem Parameter - hmtx

hmtx (HMTX) - input
The handle of the mutex semaphore to release.

DosReleaseMutexSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosReleaseMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
288	ERROR_NOT_OWNER

For a full list of error codes, see Errors.

DosReleaseMutexSem - Parameters

hmtx (HMTX) - input
The handle of the mutex semaphore to release.

ulrc (APIRET) - returns
Return Code.

DosReleaseMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
288	ERROR_NOT_OWNER

For a full list of error codes, see Errors.

DosReleaseMutexSem - Remarks

DosReleaseMutexSem relinquishes ownership of a mutex semaphore that was requested by DosRequestMutexSem.

Only the thread that owns the mutex semaphore can issue DosReleaseMutexSem.

DosReleaseMutexSem - Related Functions

Related Functions

- DosCloseMutexSem
- DosCreateMutexSem
- DosOpenMutexSem
- DosQueryMutexSem
- DosRequestMutexSem

DosReleaseMutexSem - Example Code

This example shows how to create, open, query, request, release, and close a Mutual exclusion (Mutex) semaphore.

```
#define INCL_DOSSEMAPHORES      /* Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HMTX    hmtx     = NULLHANDLE; /* Mutex semaphore handle */
PID     pidOwner = 0;          /* PID of current mutex semaphore owner */
TID     tidOwner = 0;          /* TID of current mutex semaphore owner */
ULONG   ulCount  = 0;          /* Request count for the semaphore */
APIRET  rc       = NO_ERROR;   /* Return code */

   rc = DosCreateMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                          &hmtx, 0, FALSE);       /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }
       /* This would normally be done by another unit of work */
   rc = DosOpenMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                        &hmtx);                 /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosRequestMutexSem(hmtx,                     /* Handle of semaphore */
                           (ULONG) SEM_INDEFINITE_WAIT);  /* Timeout (none) */
   if (rc != NO_ERROR) {
      printf("DosRequestMutexSem error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryMutexSem(hmtx,         /* Handle of semaphore */
                         &pidOwner,    /* Process ID of owner */
                         &tidOwner,    /* Thread ID of owner */
                         &ulCount);    /* Count */
   if (rc != NO_ERROR) {
      printf("DosQueryMutexSem error: return code = %u\n", rc);
      return 1;
   } else  {
      printf("Semaphore owned by PID %u, TID %u.", pidOwner, tidOwner);
      printf("  Request count is %u.\n", ulCount);
   } /* endif */
```

```
   rc = DosReleaseMutexSem(hmtx);        /* Relinquish ownership */
   if (rc != NO_ERROR) {
      printf("DosReleaseMutexSem error: return code = %u\n", rc);
      return 1;
    }
   rc = DosCloseMutexSem(hmtx);          /* Close mutex semaphore */
   if (rc != NO_ERROR) {
      printf("DosCloseMutexSem error: return code = %u\n", rc);
      return 1;
   }

return NO_ERROR;
}
```

DosReleaseMutexSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosRequestMutexSem

DosRequestMutexSem - Syntax

Requests ownership of a mutex semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMTX      hmtx;       /*  The handle of the mutex semaphore to request. */
ULONG     ulTimeout;  /*  The time-out in milliseconds. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosRequestMutexSem(hmtx, ulTimeout);
```

DosRequestMutexSem Parameter - hmtx

hmtx (HMTX) - input
The handle of the mutex semaphore to request.

DosRequestMutexSem Parameter - ulTimeout

ulTimeout (ULONG) - input
The time-out in milliseconds.

This is the maximum amount of time the user wants to allow the thread to be blocked.

This parameter can also have the following values:

0L	SEM_IMMEDIATE_RETURN DosRequestMutexSem returns immediately without blocking the calling thread.
-1L	SEM_INDEFINITE_WAIT DosRequestMutexSem blocks the calling thread indefinitely.

DosRequestMutexSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosRequestMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
95	ERROR_INTERRUPT
103	ERROR_TOO_MANY_SEM_REQUESTS
105	ERROR_SEM_OWNER_DIED
640	ERROR_TIMEOUT

For a full list of error codes, see Errors.

DosRequestMutexSem - Parameters

hmtx (HMTX) - input
The handle of the mutex semaphore to request.

ulTimeout (ULONG) - input
The time-out in milliseconds.

This is the maximum amount of time the user wants to allow the thread to be blocked.

This parameter can also have the following values:

0L	SEM_IMMEDIATE_RETURN DosRequestMutexSem returns immediately without blocking the calling thread.
-1L	SEM_INDEFINITE_WAIT DosRequestMutexSem blocks the calling thread indefinitely.

ulrc (APIRET) - returns
Return Code.

DosRequestMutexSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
95	ERROR_INTERRUPT

103	ERROR_TOO_MANY_SEM_REQUESTS
105	ERROR_SEM_OWNER_DIED
640	ERROR_TIMEOUT

For a full list of error codes, see Errors.

DosRequestMutexSem - Remarks

DosRequestMutexSem requests ownership of a mutex semaphore. This function can be called by any thread in the process that created the semaphore. Threads in other processes can also call this function, but they must first gain access to the semaphore by issuing DosOpenMutexSem.

DosRequestMutexSem - Related Functions

Related Functions

- DosCloseMutexSem
- DosCreateMutexSem
- DosOpenMutexSem
- DosQueryMutexSem
- DosReleaseMutexSem

DosRequestMutexSem - Example Code

This example shows how to create, open, query, request, release, and close a Mutual exclusion (Mutex) semaphore.

```
#define INCL_DOSSEMAPHORES      /* Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
HMTX    hmtx     = NULLHANDLE; /* Mutex semaphore handle */
PID     pidOwner = 0;          /* PID of current mutex semaphore owner */
TID     tidOwner = 0;          /* TID of current mutex semaphore owner */
ULONG   ulCount  = 0;          /* Request count for the semaphore */
APIRET  rc       = NO_ERROR;   /* Return code */

   rc = DosCreateMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                          &hmtx, 0, FALSE);       /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }
       /* This would normally be done by another unit of work */
   rc = DosOpenMutexSem("\\SEM32\\MUTEX1",      /* Semaphore name */
                        &hmtx);                 /* Handle returned */
   if (rc != NO_ERROR) {
      printf("DosOpenMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosRequestMutexSem(hmtx,                     /* Handle of semaphore */
                           (ULONG) SEM_INDEFINITE_WAIT);  /* Timeout (none) */
   if (rc != NO_ERROR) {
      printf("DosRequestMutexSem error: return code = %u\n", rc);
      return 1;
   }

   rc = DosQueryMutexSem(hmtx,         /* Handle of semaphore */
                         &pidOwner,    /* Process ID of owner */
```

```
                        &tidOwner,    /* Thread ID of owner */
                        &ulCount);    /* Count */
   if (rc != NO_ERROR) {
      printf("DosQueryMutexSem error: return code = %u\n", rc);
      return 1;
   } else  {
      printf("Semaphore owned by PID %u, TID %u.", pidOwner, tidOwner);
      printf("  Request count is %u.\n", ulCount);
   } /* endif */

   rc = DosReleaseMutexSem(hmtx);        /* Relinquish ownership */
   if (rc != NO_ERROR) {
      printf("DosReleaseMutexSem error: return code = %u\n", rc);
      return 1;
    }

   rc = DosCloseMutexSem(hmtx);          /* Close mutex semaphore */
   if (rc != NO_ERROR) {
      printf("DosCloseMutexSem error: return code = %u\n", rc);
      return 1;
   }

return NO_ERROR;
}
```

DosRequestMutexSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosRequestVDD

DosRequestVDD - Syntax

Allows a protected-mode OS/2 session to communicate with a virtual device driver (VDD).

```
#define INCL_DOSMVDM
#include <os2.h>

HVDD      hvdd;      /*  The handle of a VDD returned by a previous call to DosOpenVDD. */
SGID      sgid;      /*  The identifier of a specific DOS session, or null. */
ULONG     cmd;       /*  A function code that is specific to a virtual device. */
ULONG     cbInput;   /*  The length, in bytes, of the application data in pInput. */
PVOID     pInput;    /*  The address of the command-specific information. */
ULONG     cbOutput;  /*  The length, in bytes, of pOutput. */
PVOID     pOutput;   /*  The address of the buffer where the VDD returns the information for the specified comma
APIRET    ulrc;      /*  Return Code. */

ulrc = DosRequestVDD(hvdd, sgid, cmd, cbInput,
         pInput, cbOutput, pOutput);
```

DosRequestVDD Parameter - hvdd

hvdd (HVDD) - input
The handle of a VDD returned by a previous call to DosOpenVDD.

DosRequestVDD Parameter - sgid

sgid (SGID) - input
The identifier of a specific DOS session, or null.

DosRequestVDD Parameter - cmd

cmd (ULONG) - input
A function code that is specific to a virtual device.

DosRequestVDD Parameter - cbInput

cbInput (ULONG) - input
The length, in bytes, of the application data in *pInput*.

DosRequestVDD Parameter - pInput

pInput (PVOID) - input
The address of the command-specific information.

The system sends this data to the virtual device driver to process the specified command.

DosRequestVDD Parameter - cbOutput

cbOutput (ULONG) - input
The length, in bytes, of *pOutput*.

DosRequestVDD Parameter - pOutput

pOutput (PVOID) - output
The address of the buffer where the VDD returns the information for the specified command.

This information is specific to the command and the virtual device driver.

DosRequestVDD Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosRequestVDD returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
21	ERROR_NOT_READY
644	ERROR_INVALID_CALLER

For a full list of error codes, see Errors.

DosRequestVDD - Parameters

hvdd (HVDD) - input
The handle of a VDD returned by a previous call to DosOpenVDD.

sgid (SGID) - input
The identifier of a specific DOS session, or null.

cmd (ULONG) - input
A function code that is specific to a virtual device.

cbInput (ULONG) - input
The length, in bytes, of the application data in *pInput*.

pInput (PVOID) - input
The address of the command-specific information.

The system sends this data to the virtual device driver to process the specified command.

cbOutput (ULONG) - input
The length, in bytes, of *pOutput*.

pOutput (PVOID) - output
The address of the buffer where the VDD returns the information for the specified command.

This information is specific to the command and the virtual device driver.

ulrc (APIRET) - returns
Return Code.

DosRequestVDD returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
21	ERROR_NOT_READY

For a full list of error codes, see Errors.

DosRequestVDD - Remarks

The system calls every DosRequestVDD procedure registered by VDHRegisterVDD under the VDD name associated with the specified handle. This calling continues until a virtual device driver gives a return code other than VDDREQ_PASS. There is no predefined order to the calling sequence.

DosRequestVDD - Related Functions

Related Functions

- DosCloseVDD
- DosOpenVDD
- DosRequestVDD

DosRequestVDD - Example Code

The following is NOT a complete C program. It is simply intended to provide an idea of how a protected-mode OS/2 process can communicate with a virtual device driver (VDD).

This example shows a protected-mode process calling a hypothetical VDD with a request to read a string of bytes from the VDD. Assume that the session identifier of the specified DOS session has been placed into SessionID already and that the sample virtual device driver has registered the name "VDD007" with the operating system.

```
#define INCL_DOSMVDM     /* Multiple DOS sessions values */
#define INCL_DOSERRORS   /* DOS Error values */
#include <os2.h>
#include <stdio.h>

UCHAR   VDDName[10] = "VDD007";    /* Name of VDD */
HVDD    VDDHandle   = NULLHANDLE;  /* Handle of VDD */
SGID    SessionID   = 0;           /* Session identifier (should be initialized */
ULONG   Command     = 3;           /* VDD function code (hypothetical) */
APIRET  rc          = NO_ERROR;    /* Return code */
UCHAR   InputBuffer[30]  = "Your command here";  /* Command information    */
UCHAR   OutputBuffer[30] = "";                   /* Output data (returned) */

   rc = DosOpenVDD(VDDName, &VDDHandle);
   if (rc != NO_ERROR) {
      printf("VDD %s was not found.\n", rc);
      return 1;
   }

   rc = DosRequestVDD(VDDHandle,              /* Handle of VDD */
                      SessionID,              /* Session ID */
                      Command,                /* Command to send to VDD */
                      sizeof(InputBuffer),    /* Length of input */
                      InputBuffer,            /* Input buffer */
                      sizeof(OutputBuffer),   /* Length of output area */
                      OutputBuffer);          /* Output from command */
   if (rc != NO_ERROR) {
       printf("DosRequestVDD error: return code = %u\n", rc);
       return 1;
   }

   rc = DosCloseVDD(VDDHandle);               /* Close the VDD */
   if (rc != NO_ERROR) {
       printf("DosCloseVDD error: return code = %u\n", rc);
```

```
        return 1;
    }
```

DosRequestVDD - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosResetBuffer

DosResetBuffer - Syntax

Writes the buffers for the specified file to the device.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;  /*  The handle of the file whose buffers are to be written to the disk. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosResetBuffer(hFile);
```

DosResetBuffer Parameter - hFile

hFile (HFILE) - input
The handle of the file whose buffers are to be written to the disk.

If *hFile* has a value of 0xFFFFFFFF, all of the buffers for all of the file handles of the process are written to the disk.

DosResetBuffer Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosResetBuffer returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosResetBuffer - Parameters

hFile (HFILE) - input
The handle of the file whose buffers are to be written to the disk.

If *hFile* has a value of 0xFFFFFFFF, all of the buffers for all of the file handles of the process are written to the disk.

ulrc (APIRET) - returns
Return Code.

DosResetBuffer returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

For a full list of error codes, see Errors.

DosResetBuffer - Remarks

When DosResetBuffer is issued for a file handle, the contents of the file's buffers are written to the disk, and the file's directory entry is updated as if the file had been closed; however, the file remains open.

DosResetBuffer should be issued with caution. When files are on diskettes, issuing DosResetBuffer could have the undesirable effect of requiring the user to insert and remove a large number of diskettes.

Named-Pipe Considerations

Issuing DosResetBuffer for a named pipe results in an operation that is similar to forcing the buffer cache to the disk. The request blocks the calling process at one end of the pipe until all written data has been read at the other end.

DosResetBuffer - Related Functions

Related Functions

- DosClose
- DosOpen
- DosWrite

DosResetBuffer - Example Code

This example writes to a file named "DOSMAN.DAT", resets the buffer, and changes the size of the file.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   UCHAR  uchFileName[20]  = "dosman.dat",     /* Name of file */
          uchFileData[4]   = "DATA";            /* Data to write to file */
   APIRET rc             = NO_ERROR;           /* Return code */

   /* Open the file dosman.dat.  Use an existing file or create a new */
   /* one if it doesn't exist.                                        */
   rc = DosOpen(uchFileName, &hfFileHandle, &ulAction, 4L,
                FILE_ARCHIVED | FILE_NORMAL,
                OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_FLAGS_NOINHERIT | OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE, 0L);
   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   }

   rc = DosWrite (hfFileHandle, (PVOID) uchFileData,
                  sizeof (uchFileData), &ulWrote);
   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   }

   rc = DosResetBuffer (hfFileHandle);
   if (rc != NO_ERROR) {
      printf("DosResetBuffer error: return code = %u\n", rc);
      return 1;
   } /* endif */

   rc = DosSetFileSize (hfFileHandle, 8L);    /* Change file size */
   if (rc != NO_ERROR) {
      printf("DosSetFileSize error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosResetBuffer - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosResetEventSem

DosResetEventSem - Syntax

Resets an event semaphore.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HEV       hev;        /*  The handle of the event semaphore to reset. */
PULONG    pulPostCt;  /*  A pointer to a ULONG in which the event semaphore's post count is returned. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosResetEventSem(hev, pulPostCt);
```

DosResetEventSem Parameter - hev

hev (HEV) - input
The handle of the event semaphore to reset.

DosResetEventSem Parameter - pulPostCt

pulPostCt (PULONG) - output
A pointer to a ULONG in which the event semaphore's post count is returned.

The post count is the number of calls to DosPostEventSem that have been made since the last time the semaphore was in the reset state.

DosResetEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosResetEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
300	ERROR_ALREADY_RESET

For a full list of error codes, see Errors.

DosResetEventSem - Parameters

hev (HEV) - input
The handle of the event semaphore to reset.

pulPostCt (PULONG) - output
A pointer to a ULONG in which the event semaphore's post count is returned.

The post count is the number of calls to DosPostEventSem that have been made since the last time the semaphore was in the reset state.

ulrc (APIRET) - returns
Return Code.

DosResetEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
300	ERROR_ALREADY_RESET

For a full list of error codes, see Errors.

DosResetEventSem - Remarks

DosResetEventSem resets an event semaphore, causing all threads that subsequently call DosWaitEventSem to be blocked. It also returns the post count for the semaphore. The post count is the number of times that DosPostEventSem has been called since the last time the semaphore was in the reset state.

This function can be called by any thread in the process that created the semaphore. Threads in other processes can also call this function, but they must first gain access to the semaphore by calling DosOpenEventSem.

DosResetEventSem - Related Functions

Related Functions

- DosCloseEventSem
- DosCreateEventSem
- DosOpenEventSem
- DosPostEventSem
- DosQueryEventSem
- DosWaitEventSem

DosResetEventSem - Example Code

This example sets up a periodic interval timer of 2 seconds, lets it run for a while, and finally stops it.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
APIRET  rc            = NO_ERROR;           /* Return code               */
ULONG   ulPostCount   = 0;                  /* Semaphore post count      */
ULONG   i             = 0;                  /* A loop index              */

   rc = DosCreateEventSem(NULL,           /* Unnamed semaphore            */
                          &hevEvent1,     /* Handle of semaphore returned */
                          DC_SEM_SHARED,  /* Indicate a shared semaphore  */
                          FALSE);         /* Put in RESET state           */
```

```
    if (rc != NO_ERROR) {
        printf("DosCreateEventSem error: return code = %u\n", rc);
        return 1;
    }

    rc = DosStartTimer(2000L,              /* 2 second interval            */
                       (HSEM) hevEvent1,   /* Semaphore to post            */
                       &htimerEvent1);     /* Timer handler (returned)     */
    if (rc != NO_ERROR) {
        printf("DosStartTimer error: return code = %u\n", rc);
        return 1;
    }

    for (i = 1 ; i < 6 ; i++) {

      rc = DosWaitEventSem(hevEvent1,15000L); /* Wait 15 seconds for timer */
      if (rc != NO_ERROR) {
          printf("DosWaitEventSem error: return code = %u\n", rc);
          return 1;
      }

      rc = DosResetEventSem(hevEvent1,       /* Reset the semaphore         */
                            &ulPostCount);   /* And get count (should be 1) */
      if (rc != NO_ERROR) {
          printf("DosWaitEventSem error: return code = %u\n", rc);
          return 1;
      }

      printf("Iteration %u: ulPostCount = %u\n", i, ulPostCount);

    } /* for loop */
    rc = DosStopTimer(htimerEvent1);       /* Stop the timer              */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u\n", rc);
        return 1;
    }

    rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore        */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u\n", rc);
        return 1;
    }

 return NO_ERROR;
}
```

DosResetEventSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosResumeThread

DosResumeThread - Syntax

Restarts a thread that was previously stopped with DosSuspendThread.

```
#define INCL_DOSPROCESS
#include <os2.h>

TID       tid;   /*  Thread identifier of the resumed thread. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosResumeThread(tid);
```

DosResumeThread Parameter - tid

tid (TID) - input
Thread identifier of the resumed thread.

DosResumeThread Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosResumeThread returns one of the following values:

0	NO_ERROR
90	ERROR_NOT_FROZEN
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosResumeThread - Parameters

tid (TID) - input
Thread identifier of the resumed thread.

ulrc (APIRET) - returns
Return Code.

DosResumeThread returns one of the following values:

0	NO_ERROR
90	ERROR_NOT_FROZEN
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosResumeThread - Remarks

If the thread is not in a suspended state when you issue DosResumeThread for it, ERROR_NOT_FROZEN is returned.

DosResumeThread - Related Functions

Related Functions

- DosCreateThread
- DosQueryThreadContext
- DosSuspendThread

DosResumeThread - Example Code

This example creates a new thread within a process, sleeps for 1 second, suspends the thread for 5 seconds, and then waits for the thread to terminate.

Compile this example with MULTITHREAD LIBRARIES. If you are using CSet/2, use the /Gm+ switch.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#define INCL_DOSERRORS      /* DOS error values          */
#include <os2.h>
#include <stdio.h>

void _System CntThreadProc(ULONG LoopMax);  /* Count Thread */

int main(VOID) {
   TID         tidCntThread = 0;     /* ID returned for newly created thread */
   PFNTHREAD   pfnCntThread = &CntThreadProc; /* Address of thread program   */
   ULONG       ulThreadParm = 100;        /* Parameter to thread routine     */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateThread(&tidCntThread,  /* Thread ID (returned by function)  */
                        pfnCntThread,   /* Address of thread program         */
                        ulThreadParm,   /* Parameter passed to ThreadProc    */
                        CREATE_READY |  /* Thread is ready when created      */
                        STACK_SPARSE,   /* Do not pre-commit stack pages     */
                        8192L);         /* Stack size, rounded to page bdy   */
   if (rc != NO_ERROR) {
      printf("DosCreateThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (1000);  /* Sleep for a second to allow thread to run a bit */

   rc = DosSuspendThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosSuspendThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (5000);  /* Sleep 5 seconds before resuming the thread     */

   rc = DosResumeThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosResumeThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosWaitThread (&tidCntThread, DCWW_WAIT);
   if (rc != NO_ERROR) {
      printf ("DosWaitThread error : return code = %u\n", rc);
   }

   printf ("Thread has completed!\n");

   return NO_ERROR;
}
void _System CntThreadProc(ULONG LoopMax )  /* Count thread */
{
```

```
   ULONG i = 0;                 /* Loop index */

   for (i=0;i < LoopMax;i++ ) {
      printf ("%d\n", i);
   }

   return;
}
```

DosResumeThread - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosScanEnv

DosScanEnv - Syntax

Searches an environment segment for an environment variable.

```
#define INCL_DOSFILEMGR
#define INCL_DOSMISC
#include <os2.h>

PSZ       pszName;    /*  Address of the name of the environment variable. */
PSZ      *ppszValue;  /*  A pointer to the PSZ in which a pointer to the environment string is returned by the s
APIRET    ulrc;       /*  Return Code. */

ulrc = DosScanEnv(pszName, ppszValue);
```

DosScanEnv Parameter - pszName

pszName (PSZ) - input
Address of the name of the environment variable.

Do not include a trailing equal sign (=), since this is not part of the name.

DosScanEnv Parameter - ppszValue

ppszValue (PSZ *) - output
A pointer to the PSZ in which a pointer to the environment string is returned by the system.

ppszValue points to the first character of the string that is the value of the environment variable, and can be passed directly to DosSearchPath.

DosScanEnv Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosScanEnv returns the following values:

0	NO_ERROR
203	ERROR_ENVVAR_NOT_FOUND

For a full list of error codes, see Errors.

DosScanEnv - Parameters

pszName (PSZ) - input
Address of the name of the environment variable.

Do not include a trailing equal sign (=), since this is not part of the name.

ppszValue (PSZ *) - output
A pointer to the PSZ in which a pointer to the environment string is returned by the system.

ppszValue points to the first character of the string that is the value of the environment variable, and can be passed directly to DosSearchPath.

ulrc (APIRET) - returns
Return Code.

DosScanEnv returns the following values:

0	NO_ERROR
203	ERROR_ENVVAR_NOT_FOUND

For a full list of error codes, see Errors.

DosScanEnv - Remarks

Assume that the process' environment contains this statement:

```
 DPATH=c:\sysdir;c:\libdir


              pszValue points here after the
              following call to DosScanEnv:

DosScanEnv("DPATH", pszValue);
```

ppszValue points to the first character of the value of the environment variable.

DosScanEnv - Related Functions

Related Functions

- DosSearchPath

DosScanEnv - Example Code

This example scans the environment segment for the "PATH" variable and displays it. It then searches the path, plus the current directory of the "VIEW.EXE" program.

```
 #define INCL_DOS
 #define INCL_DOSERRORS       /* DOS error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

   PSZ       PathValue         = "";        /* PATH environment variable    */
   UCHAR     SearchResult[256] = "";        /* Result of PATH search        */
   APIRET    rc                = NO_ERROR;  /* Return code                  */

   rc=DosScanEnv("PATH",&PathValue);  /* Get contents of PATH environment
                                         variable                          */
   if (rc != NO_ERROR) {
       printf("DosScanEnv error: return code = %u\n",rc);
       return 1;
   } else {
       printf("PATH is:\n%s\n\n", PathValue);
   }
      /* Scan the current directory and path for the VIEW.EXE program.
         Ignore any errors from network drives which may not be in use. */

   rc=DosSearchPath(SEARCH_CUR_DIRECTORY | SEARCH_IGNORENETERRS,
                    PathValue,               /* Path value just obtained */
                    "VIEW.EXE",              /* Name of file to look for */
                    SearchResult,            /* Result of the search     */
                    sizeof(SearchResult));   /* Length of search buffer  */

   if (rc != NO_ERROR) {
       printf("DosSearchPath error: return code = %u\n",rc);
       return 1;
   } else {
       printf("Found desired file -- %s\n", SearchResult);
   }
  return NO_ERROR;
 }
```

DosScanEnv - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code

DosSearchPath

DosSearchPath - Syntax

Finds files residing along paths. The path string may come from the process environment, or be supplied directly by the caller.

```
#define INCL_DOSFILEMGR
#define INCL_DOSMISC
#include <os2.h>

ULONG     flag;           /*  A word bit vector that controls the behavior of DosSearchPath. */
PSZ       pszPathOrName;  /*  Address of the path. */
PSZ       pszFilename;    /*  Address of the ASCIIZ file name. */
PBYTE     pBuf;           /*  Address of the path name of the file, if found. */
ULONG     cbBuf;          /*  The length, in bytes, of pBuf. */
APIRET    ulrc;           /*  Return Code. */

ulrc = DosSearchPath(flag, pszPathOrName,
         pszFilename, pBuf, cbBuf);
```

DosSearchPath Parameter - flag

flag (ULONG) - input
A word bit vector that controls the behavior of DosSearchPath.

This parameter contains the following bit fields:

Bit	Description
31-3	Reserved; must be 0.
2	SEARCH_IGNORENETERRS (0x00000004) Ignore Network Errors bit. This bit controls whether the search will abort if it encounters a network error, or will continue the search with the next element. This allows you to place network paths in the PATH variable and be able to find executables in components of the PATH variable, even if the network returns an error, for example, if a server is down. If the Ignore Network Errors Bit is 0, DosSearchPath will end the search if it encounters an error from the network. If the Ignore Network Errors Bit is 1, DosSearchPath will continue the search if it encounters network errors.
1	SEARCH_ENVIRONMENT (0x00000002) Path Source bit. This bit determines how DosSearchPath interprets *pszPathOrName* 0 — *pszPathOrName* points to the actual search path. The search path string may be anywhere in the calling process's address space. Therefore, it may be in the environment, but is not required. 1 — *pszPathOrName* . points to the name of an environment variable in the process environment, and that environment variable contains the search path.
0	SEARCH_CUR_DIRECTORY (0x00000001) Implied Current bit. This bit controls whether the current directory is implicitly on the front of the

search path.

0	DosSearchPath only searches the current directory if it appears in the search path.
1	DosSearchPath searches the current working directory before it searches the directories in the search path.

For example, Implied Current bit = 0 and path = ".\;a;b" is equivalent to Implied Current bit = 1 and path = "a;b".

DosSearchPath Parameter - pszPathOrName

pszPathOrName (PSZ) - input
Address of the path.

If the Path Source bit of *flag* is 0, *pszPathOrName* is the search path that may be anywhere in the caller's address space.

If the Path Source bit of *flag* is 1, *pszPathOrName* is the name of an environment variable that contains the search path.

A search path consists of a sequence of paths separated by a semicolon (;). It is a single ASCIIZ string. The directories are searched in the order they appear in the path. Paths that contain semicolons should be quoted. For example:

```
"c:&this  is  ;  one directory path";thisisanother
```

Environment variable names are simply strings that match name strings in the environment. The equal (=) sign is not part of the name.

DosSearchPath Parameter - pszFilename

pszFilename (PSZ) - input
Address of the ASCIIZ file name.

It may contain global file-name characters. If *pszFilename* does contain global file-name characters, they remain in the result path returned in *pBuf*. This allows applications like CMD.EXE to pass the output directly to DosFindFirst. If there are no global file-name characters in *pszFilename*, the resulting path returned in *pBuf* is a fully qualified name, and may be passed directly to DosOpen, or any other system function.

DosSearchPath Parameter - pBuf

pBuf (PBYTE) - output
Address of the path name of the file, if found.

If *pszFilename* is found in one of the directories along the path, its full path name is returned in *pBuf* (with global file-name characters from *pszFilename* left in place). The contents of *pBuf* are not meaningful if DosSearchPath returns a nonzero return code.

DosSearchPath Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of *pBuf* .

DosSearchPath Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSearchPath returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
2	ERROR_FILE_NOT_FOUND
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
203	ERROR_ENVVAR_NOT_FOUND

For a full list of error codes, see Errors.

DosSearchPath - Parameters

flag (ULONG) - input
A word bit vector that controls the behavior of DosSearchPath.

This parameter contains the following bit fields:

Bit	Description
31-3	Reserved; must be 0.
2	SEARCH_IGNORENETERRS (0x00000004) Ignore Network Errors bit. This bit controls whether the search will abort if it encounters a network error, or will continue the search with the next element. This allows you to place network paths in the PATH variable and be able to find executables in components of the PATH variable, even if the network returns an error, for example, if a server is down. If the Ignore Network Errors Bit is 0, DosSearchPath will end the search if it encounters an error from the network. If the Ignore Network Errors Bit is 1, DosSearchPath will continue the search if it encounters network errors.
1	SEARCH_ENVIRONMENT (0x00000002) Path Source bit. This bit determines how DosSearchPath interprets *pszPathOrName* 0 — *pszPathOrName* points to the actual search path. The search path string may be anywhere in the calling process's address space. Therefore, it may be in the environment, but is not required. 1 — *pszPathOrName* . points to the name of an environment variable in the process environment, and that environment variable contains the search path.
0	SEARCH_CUR_DIRECTORY (0x00000001) Implied Current bit. This bit controls whether the current directory is implicitly on the front of the search path. 0 — DosSearchPath only searches the current directory if it appears in the search path. 1 — DosSearchPath searches the current working directory before it searches the directories in the search path. For example, Implied Current bit = 0 and path = ".\;a;b" is equivalent to Implied Current bit = 1 and

path = "a;b".

pszPathOrName (PSZ) - input
Address of the path.

If the Path Source bit of *flag* is 0, *pszPathOrName* is the search path that may be anywhere in the caller's address space.

If the Path Source bit of *flag* is 1, *pszPathOrName* is the name of an environment variable that contains the search path.

A search path consists of a sequence of paths separated by a semicolon (;). It is a single ASCIIZ string. The directories are searched in the order they appear in the path. Paths that contain semicolons should be quoted. For example:

```
"c:&this  is  ;  one directory path";thisisanother
```

Environment variable names are simply strings that match name strings in the environment. The equal (=) sign is not part of the name.

pszFilename (PSZ) - input
Address of the ASCIIZ file name.

It may contain global file-name characters. If *pszFilename* does contain global file-name characters, they remain in the result path returned in *pBuf*. This allows applications like CMD.EXE to pass the output directly to DosFindFirst. If there are no global file-name characters in *pszFilename*, the resulting path returned in *pBuf* is a fully qualified name, and may be passed directly to DosOpen, or any other system function.

pBuf (PBYTE) - output
Address of the path name of the file, if found.

If *pszFilename* is found in one of the directories along the path, its full path name is returned in *pBuf* (with global file-name characters from *pszFilename* left in place). The contents of *pBuf* are not meaningful if DosSearchPath returns a nonzero return code.

cbBuf (ULONG) - input
The length, in bytes, of *pBuf*.

ulrc (APIRET) - returns
Return Code.

DosSearchPath returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
2	ERROR_FILE_NOT_FOUND
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
203	ERROR_ENVVAR_NOT_FOUND

For a full list of error codes, see Errors.

DosSearchPath - Remarks

pszPathOrName always points to an ASCIIZ string. Let DPATH be an environment variable in the environment segment of the process.

```
DPATH=c:\sysdir;c:\init   /* In the environment */
```

The following two code fragments are equivalent:

```
DosScanEnv("DPATH", &PathRef);
DosSearchPath(0, /* Path Source Bit = 0 */
    PathRef, "myprog.ini", &ResultBuffer, ResultBufLen);

DosSearchPath(2, /* Path Source Bit = 1 */
    "DPATH", "myprog.ini", &ResultBuffer, ResultBufLen);
```

They both use the search path stored as DPATH in the environment segment. In the first case, the application issues DosScanEnv to find the variable; in the second case, DosSearchPath issues DosScanEnv for the application.

DosSearchPath does not check for consistency or formatting of the names. It issues DosFindFirst on a series of names that it builds from *pszPathOrName* and *pszFilename*

To determine the size of the returned path name, *pBuf* must be scanned for the ASCIIZ terminator.

An application must issue DosQuerySysInfo to determine the maximum path length that the operating system supports. The returned value should be used to dynamically allocate buffers that are to be used to store paths.

DosSearchPath - Related Functions

Related Functions

- DosFindFirst
- DosFindNext
- DosQuerySysInfo
- DosScanEnv

DosSearchPath - Example Code

This example scans the environment segment for the "PATH" variable and displays it. It then searches the path, plus the current directory of the "VIEW.EXE" program.

```
 #define INCL_DOS
 #define INCL_DOSERRORS       /* DOS error values */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

   PSZ       PathValue         = "";        /* PATH environment variable    */
   UCHAR     SearchResult[256] = "";        /* Result of PATH search        */
   APIRET    rc                = NO_ERROR;  /* Return code                  */

   rc=DosScanEnv("PATH",&PathValue);  /* Get contents of PATH environment
                                         variable                          */
   if (rc != NO_ERROR) {
       printf("DosScanEnv error: return code = %u\n",rc);
       return 1;
   } else {
       printf("PATH is:\n%s\n\n", PathValue);
   }
      /* Scan the current directory and path for the VIEW.EXE program.
         Ignore any errors from network drives which may not be in use. */

   rc=DosSearchPath(SEARCH_CUR_DIRECTORY | SEARCH_IGNORENETERRS,
                    PathValue,               /* Path value just obtained */
                    "VIEW.EXE",              /* Name of file to look for */
                    SearchResult,            /* Result of the search     */
                    sizeof(SearchResult));   /* Length of search buffer  */

   if (rc != NO_ERROR) {
       printf("DosSearchPath error: return code = %u\n",rc);
       return 1;
   } else {
       printf("Found desired file -- %s\n", SearchResult);
   }
  return NO_ERROR;
 }
```

DosSearchPath - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSelectSession

DosSelectSession - Syntax

Allows a parent session to switch one of its child sessions to the foreground.

```
#define INCL_dossesmgr
#include <os2.h>

ULONG     idSession;  /*  The identifier of the session to be switched to the foreground. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosSelectSession(idSession);
```

DosSelectSession Parameter - idSession

idSession (ULONG) - input
The identifier of the session to be switched to the foreground.

The value specified must have been returned on a previous call to DosStartSession, except that a value of zero indicates switching the caller's session (that is, the parent session) to the foreground.

DosSelectSession Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSelectSession returns one of the following values:

0	NO_ERROR
224	ERROR_SMG_NO_TARGET_WINDOW
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
459	ERROR_SMG_BAD_RESERVE
460	ERROR_SMG_PROCESS_NOT_PARENT
463	ERROR_SMG_RETRY_SUB_ALLOC

DosSelectSession - Parameters

idSession (ULONG) - input
The identifier of the session to be switched to the foreground.

The value specified must have been returned on a previous call to DosStartSession, except that a value of zero indicates switching the caller's session (that is, the parent session) to the foreground.

ulrc (APIRET) - returns
Return Code.

DosSelectSession returns one of the following values:

0	NO_ERROR
224	ERROR_SMG_NO_TARGET_WINDOW
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
459	ERROR_SMG_BAD_RESERVE
460	ERROR_SMG_PROCESS_NOT_PARENT
463	ERROR_SMG_RETRY_SUB_ALLOC

DosSelectSession - Remarks

DosSelectSession allows a parent session to switch one of its child sessions to the foreground. The session specified will not be brought to the foreground unless the parent session or one of its descendant sessions is currently executing in the foreground.

The foreground session for windowed applications is the session of the application that owns the window focus.

DosSelectSession may only be issued by a parent session to select itself or a child session. DosSelectSession may not be used to select a grandchild session, or any other descendant session beyond a child session. DosSelectSession may only be issued by the process that originally started the specified session (*idSession*) through DosStartSession.

DosSelectSession may only be used to select child sessions that were originally started by the caller with DosStartSession specifying a value of 1 for *Related*. That is, sessions started as independent sessions may not be selected through DosSelectSession.

When DosSelectSession is issued, the session specified will not be brought to the foreground unless the parent session or one of its descendant sessions is currently executing in the foreground.

Return code ERROR_SMG_NO_TARGET_WINDOW is a warning that the session might not be brought to the foreground. If the selected session is a Presentation Manager (PM) application, its window must be created with the FCF_TASKLIST flag bit set on. If the window is created with this bit set off, its session cannot be selected using DosSelectSession, and ERROR_SMG_NO_TARGET_WINDOW is returned.

If you issue DosSelectSession before creating the PM window of the selected session, ERROR_SMG_NO_TARGET_WINDOW is returned. However, if the PM window of the selected session is subsequently created with the FCF_TASKLIST flag bit set on, the window is brought to the foreground if the issuer of DosSelectSession still owns the foreground focus.

If a session still exists but its window has been destroyed, and you issue DosSelectSession for that session, ERROR_SMG_NO_TARGET_WINDOW is returned.

DosSelectSession - Related Functions

Related Functions

- DosSetSession
- DosStartSession
- DosStopSession

DosSelectSession - Example Code

This example shows how to bring a child process to the foreground.

```
#define INCL_DOSPROCESS   /* DOS process values - needed for DosSleep */
#define INCL_DOSSESMGR
#define INCL_DOSERRORS
#include <stdio.h>
#include <os2.h>

int main(VOID) {
   STARTDATA SData       = {0};
   PSZ       PgmTitle    = "PEEK-A-BOO!   I see you!",  /* Title      */
             PgmName     = "CMD.EXE";  /* This starts an OS/2 session */
   APIRET    rc          = NO_ERROR;   /* Return code                 */
   PID       pid         = 0;          /* PID returned                */
   ULONG     ulSessID    = 0;          /* Session ID returned         */
   UCHAR     achObjBuf[100] = {0};

   SData.Length  = sizeof(STARTDATA);
   SData.Related = SSF_RELATED_CHILD;       /* start a dependent session    */
   SData.FgBg    = SSF_FGBG_BACK;           /* start session in background  */
   SData.TraceOpt = SSF_TRACEOPT_NONE;      /* No trace                     */
            /* Start an OS/2 session using "CMD.EXE /K" */
   SData.PgmTitle = PgmTitle;
   SData.PgmName = PgmName;
   SData.PgmInputs = "/K";                     /* Keep session up           */
   SData.TermQ = 0;                            /* No termination queue      */
   SData.Environment = 0;                      /* No environment string     */
   SData.InheritOpt = SSF_INHERTOPT_SHELL;     /* Inherit shell's environ.  */
   SData.SessionType = SSF_TYPE_WINDOWABLEVIO; /* Windowed VIO session      */
   SData.IconFile = 0;                         /* No icon association       */
   SData.PgmHandle = 0;
   SData.PgmControl = SSF_CONTROL_VISIBLE | SSF_CONTROL_MAXIMIZE;
   SData.InitXPos  = 30;     /* Initial window coordinates              */
   SData.InitYPos  = 40;
   SData.InitXSize = 200;    /* Initial window size */
   SData.InitYSize = 140;
   SData.Reserved = 0;
   SData.ObjectBuffer  = achObjBuf; /* Contains info if DosExecPgm fails */
   SData.ObjectBuffLen = (ULONG) sizeof(achObjBuf);

   rc = DosStartSession(&SData, &ulSessID, &pid);  /* Start the session */
   if (rc != NO_ERROR) {
      printf ("DosStartSession error : return code = %u\n", rc);
      return 1;
   }

   printf("Child session will appear in the foreground in a moment.\n");
   rc = DosSleep(2500L);

   rc = DosSelectSession(ulSessID);
   if (rc != NO_ERROR) {
      printf ("DosSelectSession error : return code = %u\n", rc);
      return 1;
   }
   rc = DosSleep(5000L);
   return NO_ERROR;
}
```

DosSelectSession - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code

Related Functions
Glossary

DosSendSignalException

DosSendSignalException - Syntax

Sends a Ctrl+C or a Ctrl+Break signal exception to another process.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PID       pid;        /*  The identification of the process that is to receive the signal exception. */
ULONG     exception;  /*  The number of the signal exception to send. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosSendSignalException(pid, exception);
```

DosSendSignalException Parameter - pid

pid (PID) - input
The identification of the process that is to receive the signal exception.

DosSendSignalException Parameter - exception

exception (ULONG) - input
The number of the signal exception to send.

Possible values are shown in the following list:

1	XCPT_SIGNAL_INTR
4	XCPT_SIGNAL_BREAK

DosSendSignalException Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSendSignalException returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
156	ERROR_SIGNAL_REFUSED
205	ERROR_NO_SIGNAL_SENT
209	ERROR_INVALID_SIGNAL_NUMBER
303	ERROR_INVALID_PROCID

For a full list of error codes, see Errors.

DosSendSignalException - Parameters

pid (PID) - input
The identification of the process that is to receive the signal exception.

exception (ULONG) - input
The number of the signal exception to send.

Possible values are shown in the following list:

1	XCPT_SIGNAL_INTR
4	XCPT_SIGNAL_BREAK

ulrc (APIRET) - returns
Return Code.

DosSendSignalException returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
156	ERROR_SIGNAL_REFUSED
205	ERROR_NO_SIGNAL_SENT
209	ERROR_INVALID_SIGNAL_NUMBER
303	ERROR_INVALID_PROCID

For a full list of error codes, see Errors.

DosSendSignalException - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosSendSignalException sends either an XCPT_SIGNAL_INTR (Ctrl+C) or an XCPT_SIGNAL_BREAK (Ctrl+Break) signal exception to another process.

For a detailed list of the system exceptions, select System Exceptions.

DosSendSignalException - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosExitMustComplete
- DosRaiseException
- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler
- DosUnwindException

DosSendSignalException - Example Code

This example sends a signal exception to another process.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#define INCL_DOSEXCEPTIONS  /* DOS exception values      */
#define INCL_DOSERRORS      /* DOS error values          */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(USHORT argc, CHAR *argv[] ) {

   APIRET      rc          = NO_ERROR; /* Return code                        */
   PID         pidToIntr   = 0;        /* Interrupt this process             */

   if ( argc < 2 ) {
      printf("sendsig error:  Need to pass PID of thread to interrupt.\n");
      return 1;
   } else {
      pidToIntr = (PID) atoi ( argv[1] );
   } /* endif */

   rc = DosSendSignalException ( pidToIntr,            /* Process to interrupt */
                                 XCPT_SIGNAL_INTR );   /* Send this signal     */

   if (rc != NO_ERROR) {
      printf("DosSendSignalException error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosSendSignalException complete.\n");
   } /* endif */

   return NO_ERROR;
}
```

DosSendSignalException - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetCurrentDir

DosSetCurrentDir - Syntax

Defines the current directory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszDir;  /*  Address of the directory path name. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosSetCurrentDir(pszDir);
```

DosSetCurrentDir Parameter - pszDir

pszDir (PSZ) - input
Address of the directory path name.

The name is an ASCIIZ string.

DosSetCurrentDir Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetCurrentDir returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosSetCurrentDir - Parameters

pszDir (PSZ) - input
Address of the directory path name.

The name is an ASCIIZ string.

ulrc (APIRET) - returns
Return Code.

DosSetCurrentDir returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED

8	ERROR_NOT_ENOUGH_MEMORY
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
206	ERROR_FILENAME_EXCED_RANGE

For a full list of error codes, see Errors.

DosSetCurrentDir - Remarks

The directory path does not change if any member of the path does not exist. The current directory changes only for the requesting process.

For file-system drivers, the case of the current directory is set by *pszDir*, and not by the case of the directories on the disk. For example, if the directory "c:\bin" is created, and a *pszDir* value of "c:\bin," is specified, the current directory returned by DosQueryCurrentDir will be "c:\bin."

Programs running without the NEWFILES bit can set the current directory to a non-8.3 file-name format.

An application must issue DosQuerySysInfo to determine the maximum path length that the operating system supports. The returned value should be used to dynamically allocate buffers that are to be used to store paths.

DosSetCurrentDir - Related Functions

Related Functions

- DosQueryCurrentDir
- DosQueryCurrentDisk
- DosQuerySysInfo
- DosSetDefaultDisk

DosSetCurrentDir - Example Code

This example creates a new directory called "TEMPPROG" in the root directory, access it, and finally removes it. Some return code checking is omitted for brevity.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   UCHAR   achOrigDirName[256] = "";            /* Original directory name  */
   UCHAR   achNewDirName[256]  = "\\TEMPPROG";  /* New directory to create  */
   UCHAR   achDirName[256]     = "";          /* Directory name for queries */
   ULONG   cbDirPathLen    = 0;               /* Length of directory path   */
   PEAOP2  pEABuf       = NULL;   /* Extended Attribute buffer pointer      */
   ULONG   ulDriveNum   = 0;      /* Drive number: current=0, A=1, B=2, ... */
   APIRET  rc           = NO_ERROR;      /* Return code                      */

   cbDirPathLen = (ULONG) sizeof(achOrigDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achOrigDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Original dir. = \\%s\n", achOrigDirName);

   pEABuf = NULL;  /* Indicate no EAs are to be defined for the directory  */
   rc = DosCreateDir(achNewDirName, pEABuf);   /* Create the new directory */

   if (rc != NO_ERROR) {
```

```
      printf("DosCreateDir error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir (achNewDirName);   /* Change to new directory     */

   ulDriveNum = 0;
   cbDirPathLen = (ULONG) sizeof(achDirName);
   rc = DosQueryCurrentDir(ulDriveNum, achDirName, &cbDirPathLen);

   if (rc != NO_ERROR) {
      printf("DosQueryCurrentDir error: return code = %u\n", rc);
      return 1;
   } else printf ("Current dir.  = \\%s\n", achDirName);

   strcpy(achDirName,"\\");

   rc = DosSetCurrentDir (achDirName);     /* Change to root directory     */
   rc = DosDeleteDir (achNewDirName);      /* Delete the new directory     */

   return NO_ERROR;
}
```

DosSetCurrentDir - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetDateTime

DosSetDateTime - Syntax

Sets the current date and time.

```
#define INCL_DOSDATETIME
#include <os2.h>

PDATETIME    pdt;
APIRET       ulrc;  /*  Return Code. */

ulrc = DosSetDateTime(pdt);
```

DosSetDateTime Parameter - pdt

pdt (PDATETIME) - input
Pointer to the DATETIME data structure.

DosSetDateTime Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetDateTime returns one of the following values:

0	NO_ERROR
327	ERROR_TS_DATETIME

For a full list of error codes, see Errors.

DosSetDateTime - Parameters

pdt (PDATETIME) - input
Pointer to the DATETIME data structure.

ulrc (APIRET) - returns
Return Code.

DosSetDateTime returns one of the following values:

0	NO_ERROR
327	ERROR_TS_DATETIME

For a full list of error codes, see Errors.

DosSetDateTime - Remarks

DosSetDateTime sets the date and time that are maintained by the operating system.

The system verifies that the day is possible for the month and year (even for leap year) and that the values specified for the parameters are within their respective limits; if either of these conditions is violated, ERROR_TS_DATETIME is returned.

To get the date and time, issue DosGetDateTime.

DosSetDateTime - Related Functions

Related Functions

- DosAsyncTimer
- DosGetDateTime
- DosSleep
- DosStartTimer
- DosStopTimer

DosSetDateTime - Example Code

This example shows how to set the clock one hour ahead.

```
 #define INCL_DOSDATETIME   /* Date and time values */
 #define INCL_DOSERRORS     /* DOS error values     */
 #include <os2.h>
 #include <stdio.h>

int main(VOID) {

   DATETIME   DateTime = {0};       /* Structure to hold date/time info.   */
   APIRET     rc       = NO_ERROR;  /* Return code                         */

   rc = DosGetDateTime(&DateTime);  /* Retrieve the current date and time  */
   if (rc != NO_ERROR) {
      printf ("DosGetDateTime error : return code = %u\n", rc);
      return 1;
   }

   DateTime.hours = (UCHAR) ((BYTE) DateTime.hours + 1);    /* Set clock ahead
                                                 for Daylight Savings Time  */

   rc = DosSetDateTime(&DateTime);  /* Update the date and time            */

   if (rc!= NO_ERROR) {
      printf ("DosSetDateTime error : return code = %u\n", rc);
      return 1;
   }

   rc = DosGetDateTime(&DateTime);  /* Retrieve the date and time          */

   if (rc!= NO_ERROR) {
      printf ("DosGetDateTime error : return code = %u\n", rc);
      return 1;
   } else {
      printf("Today is %d-%d-%d; the time is now %d:%2.2d\n", DateTime.month,
               DateTime.day, DateTime.year, DateTime.hours, DateTime.minutes);
   }
   return NO_ERROR;
}
```

DosSetDateTime - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetDefaultDisk

DosSetDefaultDisk - Syntax

Sets the specified drive as the default drive.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     disknum;  /*  New default-drive number. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosSetDefaultDisk(disknum);
```

DosSetDefaultDisk Parameter - disknum

disknum (ULONG) - input
New default-drive number.

The value 1 means drive A, 2 means drive B, 3 means drive C, and so on. The maximum possible value is 26, which corresponds to drive Z. Values outside this range will cause an error.

DosSetDefaultDisk Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetDefaultDisk returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE

For a full list of error codes, see Errors.

DosSetDefaultDisk - Parameters

disknum (ULONG) - input
New default-drive number.

The value 1 means drive A, 2 means drive B, 3 means drive C, and so on. The maximum possible value is 26, which corresponds to drive Z. Values outside this range will cause an error.

ulrc (APIRET) - returns
Return Code.

DosSetDefaultDisk returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE

For a full list of error codes, see Errors.

DosSetDefaultDisk - Related Functions

Related Functions

- DosQueryCurrentDir
- DosQueryCurrentDisk
- DosSetCurrentDir

DosSetDefaultDisk - Example Code

This example creates a backup copy of the file "CONFIG.SYS" with the new name "CONFIG.CPY", in the root directory, even if the new file name already exists.

```
 #define INCL_DOSFILEMGR   /* File Manager values */
 #define INCL_DOSERRORS    /* DOS error values    */
 #include <os2.h>
 #include <stdio.h>
 #include <string.h>

int main(VOID) {
   UCHAR   achSourceString[80]  = "config.sys";  /* String to transform */
   UCHAR   achEditString[80]    = "*.cpy";       /* Editing string */
   UCHAR   achTargetString[200] = "";            /* Destination string buffer */
   APIRET  rc                   = NO_ERROR;      /* Return code */

   rc = DosSetDefaultDisk(3);    /* Set drive to C: (1=A, 2=B, 3=C, ...)      */
   if (rc != NO_ERROR) {
      printf("DosSetDefaultDisk error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSetCurrentDir ("\\");  /*  Set directory to root */
   if (rc != NO_ERROR) {
      printf("DosSetCurrentDir error: return code = %u\n", rc);
      return 1;
   }

           /* Transform "CONFIG.SYS" using "*.CPY" to "CONFIG.CPY" */

   rc = DosEditName(1, achSourceString, achEditString, achTargetString, 200);

   if (rc != NO_ERROR) {
      printf("DosEditName error: return code = %u\n", rc);
      return 1;
   }

           /* Copy contents of CONFIG.SYS to the backup file */

   rc = DosCopy(achSourceString,             /* Name of file to be copied    */
                achTargetString,             /* Name of the target file      */
                DCPY_EXISTING);   /* Copy even if target file already exists */

   if (rc != NO_ERROR) {
      printf("DosCopy error: return code = %u\n", rc);
      return 1;
   } else  printf ("Backup file %s created.\n", achTargetString);
   return NO_ERROR;
}
```

DosSetDefaultDisk - Topics

Select an item:
Syntax

Parameters
Returns
Example Code
Related Functions
Glossary

DosSetExceptionHandler

DosSetExceptionHandler - Syntax

Registers an exception handler for the current thread.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PEXCEPTIONREGISTRATIONRECORD    pERegRec;  /*  A pointer to the exception registration record that describes the
APIRET                          ulrc;      /*  Return Code. */

ulrc = DosSetExceptionHandler(pERegRec);
```

DosSetExceptionHandler Parameter - pERegRec

pERegRec (PEXCEPTIONREGISTRATIONRECORD) - input
A pointer to the exception registration record that describes the exception handler to be registered.

This exception registration record must be on the stack.

DosSetExceptionHandler Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetExceptionHandler returns one of the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosSetExceptionHandler - Parameters

pERegRec (PEXCEPTIONREGISTRATIONRECORD) - input
A pointer to the exception registration record that describes the exception handler to be registered.

This exception registration record must be on the stack.

ulrc (APIRET) - returns
Return Code.

DosSetExceptionHandler returns one of the following value:

0	NO_ERROR

For a full list of error codes, see Errors.

DosSetExceptionHandler - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosSetExceptionHandler registers an exception handler for the current thread.

If you register more than one exception handler within the same procedure, each handler's exception registration record must have a lower storage address (a higher position on the stack) than the exception registration record of the previously installed handler.

For a detailed list of the system exceptions, see System Exceptions.

DosSetExceptionHandler - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosExitMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler
- DosUnwindException

DosSetExceptionHandler - Example Code

This example shows how to set an exception handler for the current thread.

```
#define INCL_DOSPROCESS       /* DOS process values (for DosSleep) */
#define INCL_DOSEXCEPTIONS    /* DOS exception values */
#define INCL_ERRORS           /* DOS error values     */
#include <os2.h>
#include <stdio.h>

ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv );

int main (VOID)
{
  EXCEPTIONREGISTRATIONRECORD RegRec  = {0};     /* Exception Registration Record */
  APIRET      rc      = NO_ERROR;   /* Return code                   */

  /* Add MyTermHandler to this thread's chain of exception handlers */
```

```
  RegRec.ExceptionHandler = (ERR)MyTermHandler;
  rc = DosSetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosSetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Terminate this program using Ctrl-C or Ctrl-Break.\n");

  rc = DosSleep(60000L);   /* Give user plenty of time to comply... */

  rc = DosUnsetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosUnsetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }
  return NO_ERROR;
}
/*************************************************************************/
ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv )
{
  printf("*** MyTermHandler entered: ExceptionNum = %x\n",p1->ExceptionNum);
  switch( p1->ExceptionNum) {
   case XCPT_SIGNAL: {
      printf(" XCPT_SIGNAL");
      switch( p1->ExceptionInfo[0] ) {
        case XCPT_SIGNAL_INTR:     { printf("_INTR"); break; }
        case XCPT_SIGNAL_KILLPROC: { printf("_KILLPROC"); break; }
        case XCPT_SIGNAL_BREAK:    { printf("_BREAK"); break; }
        default:;
      }
      printf("\n"); break;
   }
   case XCPT_PROCESS_TERMINATE:       { printf(" Process Terminate \n"); break;}
   case XCPT_ASYNC_PROCESS_TERMINATE: { printf(" Async Process Term\n"); break;}
   case XCPT_UNWIND:                  { printf(" XCPT_UNWIND\n");        break;}
   default:;
  }

  return XCPT_CONTINUE_SEARCH;          /* Exception not resolved... */
}
```

DosSetExceptionHandler - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetExtLIBPATH

DosSetExtLIBPATH - Syntax

Defines the current path to be searched before or after the system LIBPATH when locating DLLs.

```
#define INCL_DOSMISC
#include <os2.h>

PSZ       pszExtLIBPATH;  /*  Extended LIBPATH string. */
ULONG     flags;          /*  Flag indicating when the new path is searched. */
APIRET    ulrc;           /*  Return Code. */

ulrc = DosSetExtLIBPATH(pszExtLIBPATH, flags);
```

DosSetExtLIBPATH Parameter - pszExtLIBPATH

pszExtLIBPATH (PSZ) - input
Extended LIBPATH string.

This buffer has a maximum size of 1024 bytes.

A pointer to a NULL string removes the extended LIBPATH.

DosSetExtLIBPATH Parameter - flags

flags (ULONG) - input
Flag indicating when the new path is searched.

Possible values are described in the following list:

BEGIN_LIBPATH (1)
The new path is searched before the LIBPATH.
END_LIBPATH (2)
The new path is searched after the LIBPATH.

DosSetExtLIBPATH Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetExtLIBPATH returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
161	ERROR_BAD_PATHNAME

For a full list of error codes, see Errors.

DosSetExtLIBPATH - Parameters

pszExtLIBPATH (PSZ) - input
Extended LIBPATH string.

This buffer has a maximum size of 1024 bytes.

A pointer to a NULL string removes the extended LIBPATH.

flags (ULONG) - input
Flag indicating when the new path is searched.

Possible values are described in the following list:

BEGIN_LIBPATH (1)
The new path is searched before the LIBPATH.
END_LIBPATH (2)
The new path is searched after the LIBPATH.

ulrc (APIRET) - returns
Return Code.

DosSetExtLIBPATH returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
161	ERROR_BAD_PATHNAME

For a full list of error codes, see Errors.

DosSetExtLIBPATH - Remarks

The LIBPATH string is an environment variable found in the CONFIG.SYS file consisting of a set of paths. If a fully-qualified path is not specified when a module is loaded, the system searches these paths to find the DLL.

There are two extended LIBPATH strings, BeginLIBPATH and EndLIBPATH. BeginLIBPATH is searched before the system LIBPATH, and EndLIBPATH is searched after both BeginLIBPATH and the system LIBPATH. These extended LIBPATHs can be set either from the command line using the "SET" command, or by calling DosSetExtLIBPATH. When DosSetExtLIBPATH is called, all modifications become specific to that process. Initial settings can be set for all processes in the system by setting the values in CONFIG.SYS using the "set" command.

Note: The extended LIBPATHs are not true environment variables, and do not appear when querying the environment.

Every process inherits the settings of BeginLIBPATH and EndLIBPATH from the process that starts it. If the extended library paths are initialized in CONFIG.SYS, those extended library paths become the initial settings for new processes. If a process changes BeginLIBPATH or EndLIBPATH and starts a new process, the new child process inherits the changed contents. Child processes that inherit their parent's extended LIBPATHs maintain their own copy. Modifications made by the parent after the child has been created have no effect on the children.

This function permits the use of two symbols within the path string: %BeginLIBPATH% and %EndLIBPATH%. These symbols are replaced with the current string settings for the extended library paths. For example, if BeginLIBPATH is set to

```
d:\MYDLLS;
```

and your application calls

```
DosSetExtLIBPATH("d:\TOMSDLLS;%BeginLIBPATH%;d:\JOESDLLS;", 1);
```

the resulting BeginLIBPATH is

```
d:\TOMSDLLS;d:\MYDLLS;d:\JOESDLLS;
```

The LIBPATHs strings are only searched when the specified DLL is not currently loaded. The only way to guarantee that the DLL being used is the correct version is to set the fully-qualified path in DosLoadModule. When a request is made to load a module and a path is not specified, the system searches the paths in the LIBPATH string and uses the first instance of the specified DLL it finds. If the new paths are added to the search strings, the system does not check those directories to see if a different version exists. Consequently, if two processes started from different directories use the same DLL, but different versions of that DLL exist in both directories, the version of the DLL loaded by the first process is the one used by both processes. The only way to prevent this from occurring is to specify the path to the DLL when DosLoadModule is called.

Consequently, if one process sets its BeginLIBPATH to C:\PROCESS1\DLL and loads the DLL MYAPP.DLL from that directory, and then a second process sets its BeginLIBPATH to C:\PROCESS2\DLL and there is for a different version of MYAPP.DLL in C:\PROCESS2\DLL, the second process will link to the DLL from C:\PROCESS1\DLL since it was already loaded.

Both BeginLIBPATH and EndLIBPATH can be set from the command line using the SET command. For more information about setting LIBPATHS from the command line, see the *OS/2 Command Reference* .

DosSetExtLIBPATH - Related Functions

Related Functions

- DosQueryExtLIBPATH

DosSetExtLIBPATH - Example Code

This example updates the paths to be searched before and after the system LIBPATH, when locating a DLL.

```
#define INCL_DOSMISC
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

int main(VOID) {

UCHAR   uchBeginLIBPATH[512] = "";       /* Begin LIBPATH value returned */
UCHAR   uchEndLIBPATH[512]   = "";       /* End LIBPATH value returned   */
APIRET  rc                   = NO_ERROR;    /* Return code               */

   rc = DosSetExtLIBPATH("C:\\TOOL_X\\VERS_20\\DLL",
                         BEGIN_LIBPATH);    /* Add to beginning LIBPATH */
   if (rc != NO_ERROR) {
       printf("DosSetExtLIBPATH error: return code = %u\n", rc);
       return 1;
   }

   rc = DosSetExtLIBPATH("C:\\TOOL_X\\VERS_10\\DLL",
                         END_LIBPATH);      /* Add to ending LIBPATH   */
   if (rc != NO_ERROR) {
       printf("DosSetExtLIBPATH error: return code = %u\n", rc);
       return 1;
   }

   rc = DosQueryExtLIBPATH(uchBeginLIBPATH,
                           BEGIN_LIBPATH);   /* Query the BeginLIBPATH */
   if (rc != NO_ERROR) {
       printf("DosQueryExtLIBPATH error: return code = %u\n", rc);
       return 1;
   }

   rc = DosQueryExtLIBPATH(uchEndLIBPATH,
                           END_LIBPATH);     /* Query the EndLIBPATH   */
   if (rc != NO_ERROR) {
       printf("DosQueryExtLIBPATH error: return code = %u\n", rc);
       return 1;
   }

   printf(" BeginLIBPATH = %s\n", uchBeginLIBPATH);
   printf("   EndLIBPATH = %s\n", uchEndLIBPATH);
```

```
  return NO_ERROR;
}
```

DosSetExtLIBPATH - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetFHState

DosSetFHState - Syntax

Sets the state of the specified file handle.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;  /*  File handle to be set. */
ULONG     mode;   /*  Contents of the fsOpenMode field defined in a previous DosOpen function. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosSetFHState(hFile, mode);
```

DosSetFHState Parameter - hFile

hFile (HFILE) - input
File handle to be set.

DosSetFHState Parameter - mode

mode (ULONG) - input
Contents of the *fsOpenMode* field defined in a previous DosOpen function.

This parameter contains the following bit fields:

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) This bit must be set to 0.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 Writes to the file may go through the system-buffer cache. 1 Writes to the file may go through the system-buffer cache, but the data is written (the actual file I/O operation is completed) before a synchronous-write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous-write operations. This flag bit is not inherited by child processes.
13	OPEN_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 Reported through the system critical-error handler. 1 Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands are always reported directly to the caller by way of a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. This flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (x00002000) Cache or No-Cache flag. The file is opened as follows: 0 The disk driver should place data from I/O operations into cache. 1 I/O operations to the file need not be done through the disk-driver cache. This bit is an advisory bit, and is used to advise file-system drivers and device drivers about whether the data should be cached. This bit, like the write-through bit, is a per-handle bit. This bit is not inherited by child processes.
11-8	These bits must be set to 0.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 File handle is inherited by a process created by DosExecPgm. 1 File handle is private to the current process.
6-4	These bits must be set to 0. Any other values are invalid.
3	This bit must be set to 0.
2-0	These bits must be set to 0. Any other values are invalid.

DosSetFHState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosSetFHState - Parameters

hFile (HFILE) - input
File handle to be set.

mode (ULONG) - input
Contents of the *fsOpenMode* field defined in a previous DosOpen function.

This parameter contains the following bit fields:

Bit	Description
15	OPEN_FLAGS_DASD (0x00008000) This bit must be set to 0.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag: 0 — Writes to the file may go through the system-buffer cache. 1 — Writes to the file may go through the system-buffer cache, but the data is written (the actual file I/O operation is completed) before a synchronous-write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous-write operations. This flag bit is not inherited by child processes.
13	OPEN_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows: 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands are always reported directly to the caller by way of a return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O functions. This flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (x00002000) Cache or No-Cache flag. The file is opened as follows: 0 — The disk driver should place data from I/O operations into cache. 1 — I/O operations to the file need not be done through the disk-driver cache. This bit is an advisory bit, and is used to advise file-system drivers and device drivers about whether the data should be cached. This bit, like the write-through bit, is a per-handle bit. This bit is not inherited by child processes.
11-8	These bits must be set to 0.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag: 0 — File handle is inherited by a process created by DosExecPgm. 1 — File handle is private to the current process.
6-4	These bits must be set to 0. Any other values are invalid.
3	This bit must be set to 0.
2-0	These bits must be set to 0. Any other values are invalid.

ulrc (APIRET) - returns
Return Code.

DosSetFHState returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosSetFHState - Remarks

The operating system does not guarantee the write order for multiple-sector write operations. If an application requires several sectors to be written in a specific order, the operator should issue the sectors as separate synchronous-write operations. Setting the Write-Through flag does not affect any previous write operation. That data can remain in the buffers.

When the application cannot handle a critical error that occurs, critical-error handling can be reset to the system. This is done by having DosSetFHState turn off the fail/errors bit, and then reissuing the I/O operation. The expected critical error recurs, and control is passed to the system critical-error handler. The precise time that the effect of this function is visible at the application level is unpredictable when asynchronous I/O operations are pending.

The file-handle-state bits set by this function can be queried by DosQueryFHState.

Named-Pipe Considerations

With DosSetFHState, the inheritance (I) bit and Write-Through (W) bit can be set or reset. Setting W to 1 prevents write-behind operations on remote pipes.

DosSetFHState - Related Functions

Related Functions

- DosClose
- DosDevIOCtl
- DosDupHandle
- DosExecPgm
- DosOpen
- DosQueryFHState

DosSetFHState - Example Code

This example queries the file handle state of the file "DOSQFH.DAT".

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

UCHAR       uchFileName[]   = "DOSQFH.DAT";    /* File to manipulate        */
HFILE       fhQryFile       = 0;               /* File handle from DosOpen  */
FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpen */
ULONG       FHState         = 0;                 /* File Handle State       */
APIRET      rc              = NO_ERROR;          /* Return code             */

                /* Create a file */

 rc = DosOpen(uchFileName, &fhQryFile,
              &ulOpenAction, 10L, FILE_NORMAL,
              OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
              OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L);
 if (rc != NO_ERROR) {
```

```
     printf("DosOpen error: return code = %u\n", rc);
     return 1;
  }

  rc = DosQueryFHState(fhQryFile, &FHState);
  if (rc != NO_ERROR) {
      printf("DosQueryFHState error: return code = %u\n", rc);
      return 1;
  } else printf("FHState is: %x\n", FHState);

    /*   Change state to indicate that data should not be cached */

    FHState &= 0x7F88;                  /* Turn off non-participating bits */
    rc = DosSetFHState(fhQryFile, FHState | OPEN_FLAGS_NO_CACHE);
    if (rc != NO_ERROR) {
        printf("DosSetFHState error: return code = %u\n", rc);
        return 1;
    }

    rc = DosClose(fhQryFile);
    /* Should check if (rc != NO_ERROR) here... */

    rc = DosDelete(uchFileName);      /* Delete the file */
    if (rc != NO_ERROR) {
        printf("DosDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    }

   return NO_ERROR;
}
```

DosSetFHState - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetFileInfo

DosSetFileInfo - Syntax

Sets file information.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hf;           /*  File handle. */
ULONG     ulInfoLevel;  /*  Level of file information being set. */
PVOID     pInfoBuf;     /*  Address of the storage area containing the structures for file information levels. *
ULONG     cbInfoBuf;    /*  The length, in bytes, of pInfoBuf. */
APIRET    ulrc;         /*  Return Code. */
```

```
ulrc = DosSetFileInfo(hf, ulInfoLevel, pInfoBuf,
         cbInfoBuf);
```

DosSetFileInfo Parameter - hf

hf (HFILE) - input
File handle.

DosSetFileInfo Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
Level of file information being set.

A value of 1 or 2 can be specified, as shown in the list below:

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

DosSetFileInfo Parameter - pInfoBuf

pInfoBuf (PVOID) - input
Address of the storage area containing the structures for file information levels.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASY)
pInfoBuf contains an EAOP2 data structure, and sets a series of EA name/value pairs.

Input	*pInfoBuf* is an EAOP2 data structure in which *fpFEA2List* points to a data area where the relevant FEA2LIST is to be found. *fpGEA2List* and *oError* are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area pointed to by *fpFEA2List* is also unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

DosSetFileInfo Parameter - cbInfoBuf

cbInfoBuf (ULONG) - input

The length, in bytes, of *pInfoBuf*.

DosSetFileInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetFileInfo returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFFER
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosSetFileInfo - Parameters

hf (HFILE) - input
File handle.

ulInfoLevel (ULONG) - input
Level of file information being set.

A value of 1 or 2 can be specified, as shown in the list below:

1 FIL_STANDARD
Level 1 file information

2 FIL_QUERYEASIZE
Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

pInfoBuf (PVOID) - input
Address of the storage area containing the structures for file information levels.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASY)
pInfoBuf contains an EAOP2 data structure, and sets a series of EA name/value pairs.

Input	*pInfoBuf* is an EAOP2 data structure in which *fpFEA2List* points to a data area where the relevant FEA2LIST is to be found. *fpGEA2List* and *oError* are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area pointed to by *fpFEA2List* is also unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

ulrc (APIRET) - returns

Return Code.

DosSetFileInfo returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFFER
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosSetFileInfo - Remarks

DosSetFileInfo is successful only when the file is opened for write access, and access by other processes is prevented by a deny-both sharing mode. If the file is already opened with conflicting sharing rights, any call to DosOpen will fail.

A value of 0 in the date and time components of a field does not change the field. For example, if both "last write date" and "last write time" are specified as 0 in the Level 1 information structure, then both attributes of the file are left unchanged. If either "last write date" or "last write time" are other than 0, both attributes of the file are set to the new values.

In the FAT file system, only the dates and times of the last write can be modified. Creation and last-access dates and times are not affected.

The last-modification date and time will be changed if the extended attributes are modified.

DosSetFileInfo - Related Functions

Related Functions

- DosClose
- DosEnumAttribute
- DosOpen
- DosQueryFileInfo
- DosQueryPathInfo
- DosResetBuffer
- DosSetFileSize
- DosSetPathInfo

DosSetFileInfo - Example Code

This example creates a read-only file named "DOSFDEL.DAT", and then changes the file attributes. It uses DosForceDelete to delete the file so it can not be restored using UNDELETE.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

UCHAR       uchFileName[]   = "DOSFDEL.DAT";   /* File we want to delete    */
HFILE       fhDelFile       = 0;               /* File handle from DosOpen  */
FILESTATUS3 fsts3FileInfo   = {{0}};  /* Information associated with file   */
ULONG       ulBufferSize    = sizeof(FILESTATUS3); /* File info buffer size */
```

```
 ULONG       ulOpenAction    = 0;                /* Action taken by DosOpen */
 APIRET      rc              = NO_ERROR;         /* Return code             */

                 /* Create a read-only file */

  rc = DosOpen(uchFileName, &fhDelFile,
               &ulOpenAction, 10L, FILE_READONLY,
               OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
               OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L);
  if (rc != NO_ERROR) {
     printf("DosOpen error: return code = %u\n", rc);
     return 1;       }

  rc = DosQueryFileInfo(fhDelFile, FIL_STANDARD,
                        &fsts3FileInfo, ulBufferSize);  /* Get standard info */
  if (rc != NO_ERROR) {
      printf("DosQueryFileInfo error: return code = %u\n", rc);
      return 1;
  } else { printf("File %s created read-only.\n",uchFileName); }

    fsts3FileInfo.attrFile  = FILE_NORMAL;
    rc = DosSetFileInfo(fhDelFile, FIL_STANDARD,
                        &fsts3FileInfo, ulBufferSize);
    if (rc != NO_ERROR) {
        printf("DosSetFileInfo error: return code = %u\n", rc);
        return 1;
    }

    rc = DosClose(fhDelFile);
    /* should check (rc != NO_ERROR) here... */

           /* Delete the file */

    rc = DosForceDelete(uchFileName);
    if (rc != NO_ERROR) {
        printf("DosForceDelete error: return code = %u\n", rc);
        return 1;
    } else {
        printf("File %s has been deleted.\n",uchFileName);
    } /* endif */

   return NO_ERROR;
}
```

DosSetFileInfo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetFileLocks

DosSetFileLocks - Syntax

Locks and unlocks a range of an open file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE        hFile;      /*  File handle. */
PFILELOCK    pflUnlock;  /*  Address of the structure containing the offset and length of a range to be unlocked
PFILELOCK    pflLock;    /*  Address of the structure containing the offset and length of a range to be locked.
ULONG        timeout;    /*  The maximum time, in milliseconds, that the process is to wait for the requested lo
ULONG        flags;      /*  Flags that describe the action to be taken. */
APIRET       ulrc;       /*  Return Code. */

ulrc = DosSetFileLocks(hFile, pflUnlock, pflLock,
         timeout, flags);
```

DosSetFileLocks Parameter - hFile

hFile (HFILE) - input
File handle.

DosSetFileLocks Parameter - pflUnlock

pflUnlock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be unlocked.

DosSetFileLocks Parameter - pflLock

pflLock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be locked.

DosSetFileLocks Parameter - timeout

timeout (ULONG) - input
The maximum time, in milliseconds, that the process is to wait for the requested locks.

DosSetFileLocks Parameter - flags

flags (ULONG) - input
Flags that describe the action to be taken.

This parameter has the following bit fields:

Bit	Description
31-2	Reserved flags
1	Atomic This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, an error is returned. If this bit is set to 0, then the lock may or may not occur atomically with the unlock.
0	Share This bit defines the type of access that other processes may have to the file range that is being locked. If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range. If this bit is set to 1, the current process and other processes have shared read-only access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

DosSetFileLocks Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetFileLocks returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
174	ERROR_ATOMIC_LOCK_NOT_SUPPORTED
175	ERROR_READ_LOCKS_NOT_SUPPORTED

For a full list of error codes, see Errors.

DosSetFileLocks - Parameters

hFile (HFILE) - input
File handle.

pflUnlock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be unlocked.

pflLock (PFILELOCK) - input
Address of the structure containing the offset and length of a range to be locked.

timeout (ULONG) - input
The maximum time, in milliseconds, that the process is to wait for the requested locks.

flags (ULONG) - input
Flags that describe the action to be taken.

This parameter has the following bit fields:

Bit	Description
31-2	Reserved flags
1	Atomic This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, an error is returned. If this bit is set to 0, then the lock may or may not occur atomically with the unlock.
0	Share This bit defines the type of access that other processes may have to the file range that is being locked. If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range. If this bit is set to 1, the current process and other processes have shared read-only access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

ulrc (APIRET) - returns
Return Code.

DosSetFileLocks returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
174	ERROR_ATOMIC_LOCK_NOT_SUPPORTED
175	ERROR_READ_LOCKS_NOT_SUPPORTED

For a full list of error codes, see Errors.

DosSetFileLocks - Remarks

DosSetFileLocks allows a process to lock and unlock a range in a file. The time during which a file range is locked should be short.

If the lock and unlock ranges are both zero, ERROR_LOCK_VIOLATION is returned to the caller.

If you only want to lock a file range, set the unlock file offset and the unlock range length to zero.

If you only want to unlock a file range, set the lock file offset and the lock range length to zero.

When the Atomic bit of *flags* is set to 0, and DosSetFileLocks specifies a lock operation and an unlock operation, the unlock operation occurs first, and then the lock operation is performed. If an error occurs during the unlock operation, an error code is returned and the lock operation is not performed. If an error occurs during the lock operation, an error code is returned and the unlock remains in effect if it was successful.

The lock operation is atomic when all of these conditions are met:

- The Atomic bit is set to 1 in *flags*
- The unlock range is the same as the lock range
- The process has shared access to the file range, and has requested exclusive access to it; or the process has exclusive access to the file range, and has requested shared access to it.

Some file system drivers (FSDs) may not support atomic lock operations. Versions of the operating system prior to OS/2 Version 2.00 do not support atomic lock operations. If the application receives the error code ERROR_ATOMIC_LOCK_NOT_SUPPORTED, the application

should unlock the file range and then lock it using a non-atomic operation (with the atomic bit set to 0 in *flags*). The application should also refresh its internal buffers before making any changes to the file.

If you issue DosClose to close a file with locks still in effect, the locks are released in no defined sequence.

If you end a process with a file open, and you have locks in effect in that file, the file is closed and the locks are released in no defined sequence.

The locked range can be anywhere in the logical file. Locking beyond the end of the file is not an error. A file range to be locked exclusively must first be cleared of any locked file subranges or overlapping locked file ranges.

If you repeat DosSetFileLocks for the same file handle and file range, then you duplicate access to the file range. Access to locked file ranges is not duplicated across DosExecPgm. The proper method of using locks is to attempt to lock the file range, and to examine the return value.

The following table shows the level of access granted when the accessed file range is locked with an exclusive lock or a shared lock. "Owner" refers to a process that owns the lock. "Non-owner" refers to a process that does not own the lock.

Action	Exclusive Lock	Shared Lock
Owner read	Success	Success
Non-owner read	Wait for unlock. Return error code after time-out.	Success
Owner write	Success	Wait for unlock. Return error code after time-out.
Non-owner write	Wait for unlock. Return error code after time-out.	Wait for unlock. Return error code after time-out.

If only locking is specified, DosSetFileLocks locks the specified file range using *pflLock* . If the lock operation cannot be accomplished, an error is returned, and the file range is not locked.

After the lock request is processed, a file range can be unlocked using the *pflUnlock* parameter of another DosSetFileLocks request. If unlocking cannot be accomplished, an error is returned.

Instead of denying read/write access to an entire file by specifying access and sharing modes with DosOpen requests, a process attempts to lock only the range needed for read/write access and examines the error code returned.

Once a specified file range is locked exclusively, read and write access by another process is denied until the file range is unlocked. If both unlocking and locking are specified by DosSetFileLocks, the unlocking operation is performed first, then locking is done.

DosSetFileLocks - Related Functions

Related Functions

- DosCancelLockRequest
- DosDupHandle
- DosExecPgm
- DosOpen

DosSetFileLocks - Example Code

This example opens or creates and opens a file named "FLOCK.DAT," and updates it using file locks.

```
#define INCL_DOSFILEMGR       /* File Manager values */
#define INCL_DOSERRORS        /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
int main(VOID) {

HFILE     FileHandle   = NULLHANDLE;  /* File handle */
ULONG     Action       = 0,           /* Action taken by DosOpen */
          Wrote        = 0,           /* Number of bytes written by DosWrite */
          i            = 0;           /* Loop index */
CHAR      FileData[40] = "Forty bytes of demonstration text data\r\n";
APIRET    rc           = NO_ERROR;    /* Return code */
FILELOCK  LockArea     = {0},         /* Area of file to lock */
          UnlockArea   = {0};         /* Area of file to unlock */

rc = DosOpen("flock.dat",                   /* File to open */
             &FileHandle,                   /* File handle */
             &Action,                       /* Action taken */
             4000L,                         /* File primary allocation */
             FILE_ARCHIVED,                 /* File attributes */
             FILE_OPEN | FILE_CREATE,       /* Open function type */
             OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
             0L);                           /* No extended attributes */
if (rc != NO_ERROR) {                       /* If open failed */
   printf("DosOpen error: return code = %u\n", rc);
   return 1;
}
LockArea.lOffset = 0L;              /* Start locking at beginning of file */
LockArea.lRange =  40L;             /* Use a lock range of 40 bytes       */

         /* Write 8000 bytes to the file, 40 bytes at a time */
for (i=0; i<200; ++i) {
  rc = DosSetFileLocks(FileHandle,        /* File handle   */
                       &UnlockArea,       /* Unlock previous record (if any) */
                       &LockArea,         /* Lock current record */
                       2000L,             /* Lock time-out value of 2 seconds */
                       0L);               /* Exclusive lock, not atomic */
  if (rc != NO_ERROR) {
     printf("DosSetFileLocks error: return code = %u\n", rc);
     return 1;
  }

  rc = DosWrite(FileHandle, FileData, sizeof(FileData), &Wrote);
  if (rc != NO_ERROR) {
     printf("DosWrite error: return code = %u\n", rc);
     return 1;
  }

   UnlockArea = LockArea;      /* Will unlock this record on next iteration */
   LockArea.lOffset += 40L;    /* Prepare to lock next record               */

} /* endfor - 8000 bytes written */
rc = DosClose(FileHandle);    /* Close file, this releases outstanding locks */
/* Should check if (rc != NO_ERROR) here ... */
return NO_ERROR;
}
```

DosSetFileLocks - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetFilePtr

DosSetFilePtr - Syntax

Moves the read/write pointer according to the type of move specified.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;     /*  The handle returned by a previous DosOpen function. */
LONG      ib;        /*  The signed distance (offset) to move, in bytes. */
ULONG     method;    /*  The method of moving. */
PULONG    ibActual;  /*  Address of the new pointer location. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosSetFilePtr(hFile, ib, method, ibActual);
```

DosSetFilePtr Parameter - hFile

hFile (HFILE) - input
The handle returned by a previous DosOpen function.

DosSetFilePtr Parameter - ib

ib (LONG) - input
The signed distance (offset) to move, in bytes.

DosSetFilePtr Parameter - method

method (ULONG) - input
The method of moving.

Specifies a location in the file from where the *ib* to move the read/write pointer starts. The values and their meanings are described in the following list:

0	FILE_BEGIN Move the pointer from the beginning of the file.
1	FILE_CURRENT Move the pointer from the current location of the read/write pointer.
2	FILE_END Move the pointer from the end of the file. Use this method to determine a file's size.

DosSetFilePtr Parameter - ibActual

ibActual (PULONG) - output
Address of the new pointer location.

DosSetFilePtr Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetFilePtr returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
132	ERROR_SEEK_ON_DEVICE
131	ERROR_NEGATIVE_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE

For a full list of error codes, see Errors.

DosSetFilePtr - Parameters

hFile (HFILE) - input
The handle returned by a previous DosOpen function.

ib (LONG) - input
The signed distance (offset) to move, in bytes.

method (ULONG) - input
The method of moving.

Specifies a location in the file from where the *ib* to move the read/write pointer starts. The values and their meanings are described in the following list:

0	FILE_BEGIN Move the pointer from the beginning of the file.
1	FILE_CURRENT Move the pointer from the current location of the read/write pointer.
2	FILE_END Move the pointer from the end of the file. Use this method to determine a file's size.

ibActual (PULONG) - output
Address of the new pointer location.

ulrc (APIRET) - returns
Return Code.

DosSetFilePtr returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
132	ERROR_SEEK_ON_DEVICE
131	ERROR_NEGATIVE_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE

For a full list of error codes, see Errors.

DosSetFilePtr - Remarks

The read/write pointer in a file is a signed 32-bit number. A negative value for *ib* moves the pointer backward in the file; a positive value moves it forward. DosSetFilePtr cannot be used to move to a negative position in the file.

DosSetFilePtr cannot be used for a character device or pipe.

DosSetFilePtr - Related Functions

Related Functions

- DosOpen
- DosRead
- DosSetFileSize
- DosWrite

DosSetFilePtr - Example Code

This example opens or creates and opens a file named "DOSTEST.DAT", writes to it, positions the file pointer back to the beginning of the file, reads from the file, and finally closes it.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(void) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  ulBytesRead    = 0;      /* Number of bytes read by DosRead */
   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   ULONG  ulLocal        = 0;      /* File pointer position after DosSetFilePtr */
   UCHAR  uchFileName[20]  = "dostest.dat",     /* Name of file */
          uchFileData[100] = " ";               /* Data to write to file */
   APIRET rc             = NO_ERROR;           /* Return code */

   /* Open the file test.dat.  Use an existing file or create a new */
   /* one if it doesn't exist.                                      */
   rc = DosOpen(uchFileName,                    /* File path name */
                &hfFileHandle,                  /* File handle */
                &ulAction,                      /* Action taken */
                100L,                           /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
                OPEN_ACTION_CREATE_IF_NEW |
                OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type */
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file */
                0L);                            /* No extended attribute */
   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosOpen: Action taken = %ld\n", ulAction);
   } /* endif */

   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n1...\n2...\n3\n");

   rc = DosWrite (hfFileHandle,                 /* File handle */
                  (PVOID) uchFileData,          /* String to be written */
```

```
                sizeof (uchFileData),        /* Size of string to be written */
                &ulWrote);                   /* Bytes actually written */

   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosSetFilePtr (hfFileHandle,          /* File Handle */
                       0L,                    /* Offset */
                       FILE_BEGIN,            /* Move from BOF */
                       &ulLocal);             /* New location address */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosRead (hfFileHandle,               /* File Handle */
                 uchFileData,                /* String to be read */
                 100L,                       /* Length of string to be read */
                 &ulBytesRead);              /* Bytes actually read */

   if (rc != NO_ERROR) {
      printf("DosRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosClose(hfFileHandle);               /* Close the file */

   if (rc != NO_ERROR) {
      printf("DosClose error: return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosSetFilePtr - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetFileSize

DosSetFileSize - Syntax

Changes the size of a file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;    /*  Handle of the file whose size to be changed. */
ULONG     cbSize;   /*  New size, in bytes, of the file. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosSetFileSize(hFile, cbSize);
```

DosSetFileSize Parameter - hFile

hFile (HFILE) - input
Handle of the file whose size to be changed.

DosSetFileSize Parameter - cbSize

cbSize (ULONG) - input
New size, in bytes, of the file.

DosSetFileSize Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetFileSize returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
112	ERROR_DISK_FULL

For a full list of error codes, see Errors.

DosSetFileSize - Parameters

hFile (HFILE) - input
Handle of the file whose size to be changed.

cbSize (ULONG) - input
New size, in bytes, of the file.

ulrc (APIRET) - returns
Return Code.

DosSetFileSize returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
112	ERROR_DISK_FULL

For a full list of error codes, see Errors.

DosSetFileSize - Remarks

When DosSetFileSize is issued, the file must be open in a mode that allows write access.

The size of the open file can be truncated or extended. If the file size is being extended, the file system tries to allocate additional bytes in a contiguous (or nearly contiguous) space on the medium. The values of the new bytes are undefined.

DosSetFileSize - Related Functions

Related Functions

- DosOpen
- DosQueryFileInfo
- DosQueryPathInfo

DosSetFileSize - Example Code

This example writes to a file named "DOSMAN.DAT", resets the buffer, and changes the file size.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   UCHAR  uchFileName[20]  = "dosman.dat",     /* Name of file */
          uchFileData[4]   = "DATA";            /* Data to write to file */
   APIRET rc             = NO_ERROR;           /* Return code */

   /* Open the file dosman.dat.  Use an existing file or create a new */
   /* one if it doesn't exist.                                        */
   rc = DosOpen(uchFileName, &hfFileHandle, &ulAction, 4L,
                FILE_ARCHIVED | FILE_NORMAL,
                OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_FLAGS_NOINHERIT | OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE, 0L);
   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   }

   rc = DosWrite (hfFileHandle, (PVOID) uchFileData,
                  sizeof (uchFileData), &ulWrote);
   if (rc != NO_ERROR) {
```

```
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   }

   rc = DosResetBuffer (hfFileHandle);
   if (rc != NO_ERROR) {
      printf("DosResetBuffer error: return code = %u\n", rc);
      return 1;
   } /* endif */

   rc = DosSetFileSize (hfFileHandle, 8L);    /* Change file size */
   if (rc != NO_ERROR) {
      printf("DosSetFileSize error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosSetFileSize - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetFSInfo

DosSetFSInfo - Syntax

Sets information for a file system device.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     disknum;
ULONG     infolevel;
PVOID     pBuf;
ULONG     cbBuf;
APIRET    ulrc;       /*  Return Code. */

ulrc = DosSetFSInfo(disknum, infolevel, pBuf,
         cbBuf);
```

DosSetFSInfo Parameter - disknum

disknum (ULONG) - input
Logical drive number. Zero means the default drive, 1 means drive A, 2 means drive B, 3 means drive C, and so on. This represents the file system driver (FSD) for the media currently in that drive. A value of 0xFFFF means that *pBuf* contains the ASCIIZ path name of the FSD.

DosSetFSInfo Parameter - infolevel

infolevel (ULONG) - input
Level of file information to be set. Only a value of 2 (FSIL_VOLSER) may be specified.

DosSetFSInfo Parameter - pBuf

pBuf (PVOID) - input
Address of the storage area where the system gets the new file system information.

Level 2 Information
Level 2 information is returned in the format of an FSINFO structure.

DosSetFSInfo Parameter - cbBuf

cbBuf (ULONG) - input
The length, in bytes, of *pBuf*.

DosSetFSInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetFSInfo returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
82	ERROR_CANNOT_MAKE
122	ERROR_INSUFFICIENT_BUFFER
123	ERROR_INVALID_NAME
124	ERROR_INVALID_LEVEL
154	ERROR_LABEL_TOO_LONG

For a full list of error codes, see Errors.

DosSetFSInfo - Parameters

disknum (ULONG) - input
Logical drive number. Zero means the default drive, 1 means drive A, 2 means drive B, 3 means drive C, and so on. This represents the file system driver (FSD) for the media currently in that drive. A value of 0xFFFF means that *pBuf* contains the ASCIIZ path name of the FSD.

infolevel (ULONG) - input
Level of file information to be set. Only a value of 2 (FSIL_VOLSER) may be specified.

pBuf (PVOID) - input
Address of the storage area where the system gets the new file system information.

Level 2 Information
Level 2 information is returned in the format of an FSINFO structure.

cbBuf (ULONG) - input
The length, in bytes, of *pBuf*.

ulrc (APIRET) - returns
Return Code.

DosSetFSInfo returns one of the following values:

0	NO_ERROR
15	ERROR_INVALID_DRIVE
82	ERROR_CANNOT_MAKE
122	ERROR_INSUFFICIENT_BUFFER
123	ERROR_INVALID_NAME
124	ERROR_INVALID_LEVEL
154	ERROR_LABEL_TOO_LONG

For a full list of error codes, see Errors.

DosSetFSInfo - Remarks

Trailing blanks supplied at the time the volume label is defined are not returned by DosQueryFSInfo.

File-system information can be set only if the volume is opened in a mode that allows write access.

DosSetFSInfo - Related Functions

Related Functions

- DosQueryCurrentDisk
- DosQueryFSInfo
- DosQuerySysInfo
- DosSetDefaultDisk

DosSetFSInfo - Example Code

This example shows how to label the disk in drive "A" as "MYDISK". Before running this program, make sure that there is a disk in the drive.

```
#define INCL_DOSFILEMGR         /* File Manager values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
int main(VOID) {

ULONG   DriveNumber   = 1;                    /* Drive 1=A: 2=B: 3=C: ... */
VOLUMELABEL FSInfoBuf = {0};                  /* File system info buffer */
APIRET  rc            = NO_ERROR;             /* Return code */

strcpy(FSInfoBuf.szVolLabel, "MYDISK");    /* Change vol label to MYDISK */
FSInfoBuf.cch = (BYTE) strlen(FSInfoBuf.szVolLabel);

rc = DosSetFSInfo(DriveNumber,            /* Drive number */
                  FSIL_VOLSER,            /* Level of information being set */
                  &FSInfoBuf,             /* Address of input buffer */
                  sizeof(VOLUMELABEL) );  /* Buffer size */
if (rc != NO_ERROR) {
   printf("DosSetFSInfo error: return code = %u\n", rc);
   return 1;
}

return NO_ERROR;
}
```

DosSetFSInfo - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetMaxFH

DosSetMaxFH - Syntax

Defines the maximum number of file handles for the calling process.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     cFH;   /*  Total number of file handles to be provided. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosSetMaxFH(cFH);
```

DosSetMaxFH Parameter - cFH

cFH (ULONG) - input
Total number of file handles to be provided.

DosSetMaxFH Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetMaxFH returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosSetMaxFH - Parameters

cFH (ULONG) - input
Total number of file handles to be provided.

ulrc (APIRET) - returns
Return Code.

DosSetMaxFH returns one of the following values:

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosSetMaxFH - Remarks

OS/2 initially allocates 50 file handles to a process. This is the recommended number for an application. However, if the system limit has not been reached, this amount can be increased with DosSetMaxFH. When DosSetMaxFH is issued, all open file handles are preserved.

If an application tries to set the number of file handles using DosSetMaxFH to a value less than the number of file handles set by OS/2 or a parent process it will return an error return code of 87, ERROR_INVALID_PARAMETER. If a higher limit is set, it can be lowered, but not lower than the value inherited from the system or the parent process.

Applications should instead use DosSetRelMaxFH, to query first and then to set, the maximum number of file handles for a process.

DosSetMaxFH - Related Functions

Related Functions

- DosDupHandle
- DosOpen
- DosSetRelMaxFH

DosSetMaxFH - Example Code

This example defines and adjust the maximum number of file handles.

```
#define INCL_DOSFILEMGR      /* File Manager values */
#define INCL_DOSERRORS       /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID){

  ULONG    CurMaxFH      = 0;          /* Current count of handles         */
  LONG     ReqCount      = 0;          /* Number to adjust file handles    */
  APIRET   rc            = NO_ERROR;   /* Return code                      */

  rc = DosSetRelMaxFH(&ReqCount,     /* Using 0 here will return the       */
                      &CurMaxFH);    /* current number of file handles     */

  if (rc != NO_ERROR) {
     printf("DosSetRelMaxFH error: return code = %u\n", rc);
     return 1;
  } else {
     printf("Maximum number of file handles is %u.\n", CurMaxFH);
  }

  rc = DosSetMaxFH(110L);            /* Set maximum file handles to 110  */

  if (rc != NO_ERROR) {
     printf("DosSetMaxFH error: return code = %u\n", rc);
     return 1;
  }

  ReqCount = -5L;                    /* Want 5 less file handles         */

  rc = DosSetRelMaxFH(&ReqCount,&CurMaxFH);     /* Change handle maximum */

  if (rc != NO_ERROR) {
     printf("DosSetRelMaxFH error: return code = %u\n", rc);
     return 1;
  } else {
     printf("Maximum number of file handles is now %u.\n", CurMaxFH);
  }

  return NO_ERROR;
}
```

DosSetMaxFH - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetMem

DosSetMem - Syntax

Commits or decommits a range of pages within a memory object, or alters their access protection.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pb;    /*  The base address of the range of pages whose attributes are to be changed. */
ULONG     cb;    /*  A value specifying the size, in bytes, of the region whose attributes are to be changed. */
ULONG     flag;  /*  A set of flags specifying commitment or decommitment, and desired access protection, for th
APIRET    ulrc;  /*  Return Code. */

ulrc = DosSetMem(pb, cb, flag);
```

DosSetMem Parameter - pb

pb (PVOID) - input
The base address of the range of pages whose attributes are to be changed.

DosSetMem Parameter - cb

cb (ULONG) - input
A value specifying the size, in bytes, of the region whose attributes are to be changed.

The size is rounded up to include all pages addressed by the requested base address and size.

DosSetMem Parameter - flag

flag (ULONG) - input
A set of flags specifying commitment or decommitment, and desired access protection, for the specified range of pages.

Commit Type

PAG_COMMIT bit (0x00000010)
The specified range of pages is to be committed.

PAG_DECOMMIT bit (0x00000020)
The specified range of pages is to be decommitted.

Note: Only committed pages in private memory objects can be decommitted. A committed page in a shared memory object cannot be decommitted. If neither is specified, no change in commitment is made.

Desired Access Protection

PAG_EXECUTE bit (0x00000004)
Execute access to the committed range of pages is desired.

PAG_READ bit (0x00000001)

Read access to the committed range of pages is desired.

PAG_WRITE bit (0x00000002)
Write access to the committed range of pages is desired.

PAG_GUARD bit (0x00000008)
Access to the committed range of pages causes a "guard page entered" condition to be raised in the subject process.

PAG_DEFAULT bit (0x00000400)
The access protection assigned to the committed range of pages is the access protection specified when the object was allocated in the address space of the requesting process.

If the PAG_DECOMMIT bit is not set, then the PAG_DEFAULT bit or at least one of the bits PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

All other bits must be clear.

DosSetMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
212	ERROR_LOCKED
487	ERROR_INVALID_ADDRESS
32798	ERROR_CROSSES_OBJECT_BOUNDARY

For a full list of error codes, see Errors.

DosSetMem - Parameters

pb (PVOID) - input
The base address of the range of pages whose attributes are to be changed.

cb (ULONG) - input
A value specifying the size, in bytes, of the region whose attributes are to be changed.

The size is rounded up to include all pages addressed by the requested base address and size.

flag (ULONG) - input
A set of flags specifying commitment or decommitment, and desired access protection, for the specified range of pages.

Commit Type

PAG_COMMIT bit (0x00000010)
The specified range of pages is to be committed.

PAG_DECOMMIT bit (0x00000020)
The specified range of pages is to be decommitted.

Note: Only committed pages in private memory objects can be decommitted. A committed page in a shared memory object cannot be decommitted. If neither is specified, no change in commitment is made.

Desired Access Protection

PAG_EXECUTE bit (0x00000004)
Execute access to the committed range of pages is desired.

PAG_READ bit (0x00000001)
Read access to the committed range of pages is desired.

PAG_WRITE bit (0x00000002)
Write access to the committed range of pages is desired.

PAG_GUARD bit (0x00000008)
Access to the committed range of pages causes a "guard page entered" condition to be raised in the subject process.

PAG_DEFAULT bit (0x00000400)
The access protection assigned to the committed range of pages is the access protection specified when the object was allocated in the address space of the requesting process.

If the PAG_DECOMMIT bit is not set, then the PAG_DEFAULT bit or at least one of the bits PAG_READ, PAG_WRITE, or PAG_EXECUTE must be specified.

All other bits must be clear.

ulrc (APIRET) - returns
Return Code.

DosSetMem returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
212	ERROR_LOCKED
487	ERROR_INVALID_ADDRESS
32798	ERROR_CROSSES_OBJECT_BOUNDARY

For a full list of error codes, see Errors.

DosSetMem - Remarks

DosSetMem can be used to commit or decommit a range of previously allocated pages in either a private or shared memory object. It also can be used to create a sparse population of committed private or shared pages within a memory object. DosSetMem can also change the access protection applied to already-committed pages within a memory object.

Each page in the virtual-address space of the process is either free, private, or shared.

The virtual address for free pages is not reserved, not committed, and not accessible. An attempt to commit or decommit a free page results in the return of an error.

The virtual address for pages in a private or shared memory object is reserved during the allocation of the memory object. Each page within a memory object can be in one of two states:

Committed	These pages have allocated backing storage, with access controlled by a protection code. A committed page in a private memory object may be decommitted; a committed page in a shared memory object may not be decommitted. An attempt to commit a previously committed page results in the return of an error.
Decommitted	These pages are not committed and are not accessible. A decommitted page may be committed if the backing storage is available. An attempt to decommit a previously decommitted page results in the return of an error.

The commitment of a reserved page in a shared object causes the page to be committed in the context of each process sharing the shared memory object.

Any access protection can be applied to committed private pages. Decommitted pages are given an access protection of "no access".

When pages are committed, they are backed by demand pages. The first attempt to read or write the page causes a page of zeros to be created.

Decommitting a private page causes the backing storage for the page to be released.

Setting the protection on a range of previously committed pages causes the old access protection to be replaced by the desired access protection. The access protection can be set only on committed pages.

Setting the access protection to PAG_GUARD causes a range of guard pages to be established. If access to this range of pages is attempted, an access violation (page fault) is generated. This fault sets the protection of the accessed page to the desired access protection, and generates a condition that signifies that a guard page has been entered. This capability is intended to provide automatic stack checking. It can also be used to separate other data structures when appropriate.

If a failure occurs, the attributes are not changed on any pages, and an appropriate error code is returned.

As each page is considered for protection, its state is determined. If the state of the page is not committed, or is not being committed, an appropriate error code is returned. Otherwise, the new protection of the page is set.

With the Intel 80386 processor, execute and read access are equivalent. Also, write access implies both read and execute access.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosSetMem - Related Functions

Related Functions

- DosAllocMem
- DosAllocSharedMem
- DosQueryMem

DosSetMem - Example Code

This example allocates, commits, queries, uses, and then frees read/write memory. DosSetMem is used to commit the memory object since it was not done at allocation time.

```
#define INCL_DOSMEMMGR   /* Include DOS Memory Management APIs */
#define INCL_DOSERRORS   /* DOS error values                   */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main (VOID)
{
   PVOID  MyObject    = NULL;      /* Pointer to memory object             */
   ULONG  ulObjSize   = 0;         /* Size of memory object (in bytes)     */
   ULONG  ulMemFlags  = 0;         /* Attribute flags for the object       */
   ULONG  ulMemSize   = 0;         /* Size of memory region for DosSetMem  */
   APIRET rc          = NO_ERROR;  /* Return code                          */

   ulObjSize = 2000;          /* Will be rounded to a page boundary - 4096 */

   rc = DosAllocMem(&MyObject,     /* Pointer to memory object pointer     */
                    ulObjSize,     /* Size of object to be allocated       */
                    PAG_WRITE );   /* Allocate memory as read/writeable    */
   if (rc != NO_ERROR) {
```

```
      printf("DosAllocMem error:  return code = %u\n",rc);
      return 1;
   }
       /* Object can't be used until it is COMMITTED.  Since this was
          not done at DosAllocMem time, do it now.                    */

   rc = DosSetMem(MyObject,                    /* Pointer to object      */
                  ulObjSize,                   /* Size of area to change */
                  PAG_DEFAULT | PAG_COMMIT );  /* Commit the object      */

   if (rc != NO_ERROR) {
      printf("DosSetMem error:  return code = %u\n",rc);
      rc = DosFreeMem(MyObject); /* If omitted, OS/2 frees it at termination */
      return 1;
   } else { printf("DosSetMem: complete\n"); }

   strcpy(MyObject, "The memory object has just been used.");

              /* Check COMMIT status of the memory object. */

   ulMemSize = ulObjSize;
   rc = DosQueryMem(MyObject, &ulMemSize, &ulMemFlags);
   if (rc == NO_ERROR) {
      if (ulMemFlags & PAG_COMMIT) {
         printf("   Page containing MyObject is now committed.\n");
      } else {
         printf("Error:  Page containing MyObject has not been committed.\n");
      } /* endif */
   } else {
      printf("DosQueryMem error: return code = %u\n",rc);
   } /* endif */
   rc = DosFreeMem(MyObject);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error:  return code = %u\n",rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosSetMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetNPHState

DosSetNPHState - Syntax

Resets the blocking mode and the read mode of a named pipe.

```
#define INCL_DOSNMPIPES
```

```
#include <os2.h>

HPIPE     hpipe;  /*  The named-pipe handle to reset. */
ULONG     state;  /*  The named-pipe handle state. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosSetNPHState(hpipe, state);
```

DosSetNPHState Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to reset.

The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen.

DosSetNPHState Parameter - state

state (ULONG) - input
The named-pipe handle state.

This parameter consists of the following bit fields:

Bit	Description
31-16	Reserved.
15	Blocking mode. The blocking mode is defined as either "blocking" or "nonblocking," as follows: 0 NP_WAIT (0x0000) Blocking mode: DosRead and DosWrite wait if no data is available. 1 NP_NOWAIT (0x8000) Nonblocking mode: DosRead and DosWrite return immediately if no data is available. DosRead normally blocks (waits) until at least partial data can be returned. DosWrite blocks by default until all of the requested bytes have been written. Nonblocking mode changes this behavior as follows: DosRead returns immediately with a value of zero for *pcbActual* if no data is available. DosWrite returns a value of zero for *pcbActual* if there is not enough buffer space available in the pipe; otherwise, the entire data area is transferred.
14-10	Reserved.
9-8	Read Mode. The read mode is defined as follows: 00 NP_READMODE_BYTE (0x0000) Byte-read mode: Read the pipe as a byte stream. 01 NP_READMODE_MESSAGE (0x0100) Message-stream mode: Read the pipe as a message stream.
7-0	Reserved, must be set to 0.

DosSetNPHState Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetNPHState returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosSetNPHState - Parameters

hpipe (HPIPE) - input
The named-pipe handle to reset.

The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen.

state (ULONG) - input
The named-pipe handle state.

This parameter consists of the following bit fields:

Bit	Description
31-16	Reserved.
15	Blocking mode. The blocking mode is defined as either "blocking" or "nonblocking," as follows: 0 NP_WAIT (0x0000) Blocking mode: DosRead and DosWrite wait if no data is available. 1 NP_NOWAIT (0x8000) Nonblocking mode: DosRead and DosWrite return immediately if no data is available. DosRead normally blocks (waits) until at least partial data can be returned. DosWrite blocks by default until all of the requested bytes have been written. Nonblocking mode changes this behavior as follows: DosRead returns immediately with a value of zero for *pcbActual* if no data is available. DosWrite returns a value of zero for *pcbActual* if there is not enough buffer space available in the pipe; otherwise, the entire data area is transferred.
14-10	Reserved.
9-8	Read Mode. The read mode is defined as follows: 00 NP_READMODE_BYTE (0x0000) Byte-read mode: Read the pipe as a byte stream. 01 NP_READMODE_MESSAGE (0x0100) Message-stream mode: Read the pipe as a message stream.
7-0	Reserved, must be set to 0.

ulrc (APIRET) - returns
Return Code.

DosSetNPHState returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosSetNPHState - Remarks

DosSetNPHState resets the blocking mode and the read mode of a named pipe. Both the blocking mode and the read mode must be specified. However, the read mode cannot be changed if the pipe is a byte pipe. (Byte pipes can be read only as byte streams.) In addition, the blocking mode cannot be changed to nonblocking if another thread is currently blocked on an I/O request to the same end of the pipe.

DosSetNPHState - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPipeSem
- DosTransactNPipe
- DosWaitNPipe
- DosWrite

DosSetNPHState - Example Code

This example handles the client side of a pipe. It opens an existing named pipe, sets and queries the pipe handle state, sends a message to the host, reads the host reply, and finally closes the pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR         /* DOS File Manager values */
#define INCL_DOSNMPIPES         /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES      /* DOS Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
```

```
   HFILE    PipeHandle             = NULLHANDLE; /* Pipe handle */
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]            = {0};
   ULONG    bytes                  = 0;
   ULONG    Action                 = 0;
   ULONG    PipeState              = 0;
   ULONG    HandType               = 0;
   ULONG    FlagWord               = 0;
   ULONG    BytesRead              = 0;

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   } else printf("Connected to HOST\n");

   rc = DosSetNPHState(PipeHandle, NP_WAIT);
   if (rc != NO_ERROR) {
      printf("DosSetNPHState error: error code = %u\n", rc);
      return 1;
   }

   rc = DosQueryNPHState(PipeHandle, &PipeState);
   if (rc != NO_ERROR) {
      printf("DosQueryNPHState error: error code = %u\n", rc);
      return 1;
   } else {
      printf("Pipe handle state is: %x\n", PipeState);
   }  /* endif */
   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);  /* Force above printf prompt to display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }

   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST was: %s\n\n", message);

   rc = DosClose(PipeHandle);
   /* Should check if (RC == NO_ERROR)  here... */

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosSetNPHState - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetNPipeSem

DosSetNPipeSem - Syntax

Attaches a shared event semaphore to a local named pipe.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE     hpipe;  /*  The named-pipe handle to which a semaphore is to be attached. */
HSEM      hsem;   /*  The handle of an event semaphore or a multiple-wait (muxwait) semaphore that is posted whe
ULONG     key;    /*  A key value that distinguishes events arriving on different named pipes that are attached
APIRET    ulrc;   /*  Return Code. */

ulrc = DosSetNPipeSem(hpipe, hsem, key);
```

DosSetNPipeSem Parameter - hpipe

hpipe (HPIPE) - input
The named-pipe handle to which a semaphore is to be attached.

The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen.

DosSetNPipeSem Parameter - hsem

hsem (HSEM) - input
The handle of an event semaphore or a multiple-wait (muxwait) semaphore that is posted when the pipe (*hpipe*) has either data to be read or write space available.

DosSetNPipeSem Parameter - key

key (ULONG) - input
A key value that distinguishes events arriving on different named pipes that are attached to the same semaphore.

DosSetNPipeSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetNPipeSem returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
187	ERROR_SEM_NOT_FOUND
230	ERROR_BAD_PIPE
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosSetNPipeSem - Parameters

hpipe (HPIPE) - input
The named-pipe handle to which a semaphore is to be attached.

The server handle is returned by DosCreateNPipe; the client handle is returned by DosOpen.

hsem (HSEM) - input
The handle of an event semaphore or a multiple-wait (muxwait) semaphore that is posted when the pipe (*hpipe*) has either data to be read or write space available.

key (ULONG) - input
A key value that distinguishes events arriving on different named pipes that are attached to the same semaphore.

ulrc (APIRET) - returns
Return Code.

DosSetNPipeSem returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
187	ERROR_SEM_NOT_FOUND
230	ERROR_BAD_PIPE
233	ERROR_PIPE_NOT_CONNECTED

For a full list of error codes, see Errors.

DosSetNPipeSem - Remarks

DosSetNPipeSem can be used with local pipes or on the client end of a remote pipe. (A remote pipe is the client end of a pipe created on a remote named pipe server.) If an attempt is made to attach a semaphore to the server end of a remote pipe, DosSetNPipeSem returns 1 (ERROR_INVALID_FUNCTION).

If a semaphore is already attached to the specified handle, DosSetNPipeSem replaces the existing semaphore with the new one.

DosSetNPipeSem - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosCloseMuxWaitSem
- DosConnectNPipe
- DosCreateEventSem

- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosTransactNPipe
- DosWaitNPipe
- DosWaitEventSem
- DosWaitMuxWaitSem
- DosWrite

DosSetNPipeSem - Example Code

This example handles the host end of a named pipe for several other named pipe examples. Some return code checking has been omitted for brevity.

```
#define INCL_BASE
#define INCL_DOSSEMAPHORES
#define INCL_DOSNMPIPES
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
CHAR     PipeName[256] = "\\PIPE\\EXAMPLE" ;     /* Pipe name */
HPIPE    PipeHandle    = NULLHANDLE;             /* Pipe handle */
HEV      hev           = NULLHANDLE;             /* Semaphore handle */
ULONG    ulBytes       = 0;                      /* Bytes read or written */
CHAR     message[256]  = "";                     /* Input/Output buffer */
APIRET   rc            = NO_ERROR;               /* Return code */

   rc = DosCreateNPipe(PipeName,               /* Name of pipe to create */
                       &PipeHandle,            /* Handle returned for pipe */
                       NP_ACCESS_DUPLEX,       /* Duplex pipe */
                       NP_WAIT |
                       NP_TYPE_MESSAGE |
                       NP_READMODE_MESSAGE |
                       NP_WMESG |              /* Write messages */
                       NP_RMESG |              /* Read messages */
                       0x01,                   /* Unique instance of pipe */
                       sizeof(message),        /* Output buffer size */
                       sizeof(message),        /* Input buffer size */
                       0L);                    /* Use default time-out */
   if (rc != NO_ERROR) {
     printf("DosCreateNPipe error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCreateEventSem("\\SEM32\\PIPE\\EXAMPLE", &hev, 0L, 0L);
   /* Should check if (rc != NO_ERROR)  here... This semaphore is not
      always used.                                                    */

   rc = DosSetNPipeSem(PipeHandle,       /* Handle for pipe */
                       (HSEM) hev,       /* Handle of semaphore */
                       1L);              /* Used to distinguish among events */
   if (rc != NO_ERROR) {
     printf("DosSetNPipeSem error: return code = %u\n",rc);
     return 1;
   }

   printf("Waiting for connection to pipe %s...\n",PipeName);

   rc = DosConnectNPipe(PipeHandle);
   if (rc != NO_ERROR) {
     printf("DosConnectNPipe error: return code = %u\n",rc);
     return 1;
   }
```

```
   printf("\nCONNECTED\nWaiting for a message...\n");
   rc = DosRead(PipeHandle,            /* Handle of pipe */
                message,               /* Buffer for message read */
                sizeof(message),       /* Buffer size */
                &ulBytes);             /* Number of bytes actually read */
   if (rc != NO_ERROR) {
     printf("DosRead error: return code = %u\n",rc);
     return 1;
   }

   printf("\n\nMessage received was: %s\n\n", message);

   strcpy(message, "Thank you for your message!");
   rc = DosWrite(PipeHandle,           /* Handle of pipe */
                 message,              /* Buffer containing message to write */
                 strlen(message),      /* Length of message */
                 &ulBytes);            /* Number of bytes actually written */
   if (rc != NO_ERROR) {
     printf("DosWrite error: return code = %u\n",rc);
     return 1;
   }

   rc = DosCloseEventSem(hev);
   /* Should check if (rc != NO_ERROR)  here... */

   rc = DosDisConnectNPipe(PipeHandle);
   /* Should check if (rc != NO_ERROR)  here... */

   return NO_ERROR;
}
```

DosSetNPipeSem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetPathInfo

DosSetPathInfo - Syntax

Sets information for a file or directory.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PSZ       pszPathName;  /*  Address of the ASCIIZ full path name of the file or subdirectory. */
ULONG     ulInfoLevel;  /*  The level of file directory information being defined. */
PVOID     pInfoBuf;     /*  Address of the storage area containing the file information being set. */
ULONG     cbInfoBuf;    /*  The length, in bytes, of pInfoBuf. */
ULONG     flOptions;    /*  Information on how the set operation is to be performed. */
APIRET    ulrc;         /*  Return Code. */
```

```
ulrc = DosSetPathInfo(pszPathName, ulInfoLevel,
         pInfoBuf, cbInfoBuf, flOptions);
```

DosSetPathInfo Parameter - pszPathName

pszPathName (PSZ) - input
Address of the ASCIIZ full path name of the file or subdirectory.

Global file-name characters are not permitted.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

DosSetPathInfo Parameter - ulInfoLevel

ulInfoLevel (ULONG) - input
The level of file directory information being defined.

A value of 1 or 2 can be specified, as shown in the following list.

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

DosSetPathInfo Parameter - pInfoBuf

pInfoBuf (PVOID) - input
Address of the storage area containing the file information being set.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure.

Level 2 sets a series of extended attribute (EA) name/value pairs.

Input	*pInfoBuf* contains an EAOP2 data structure. *fpGEA2List* is ignored. *fpFEA2List* points to a data area where the relevant FEA2 list is to be found. *oError* is ignored. The FEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the FEA2 list. The *oNextEntryOffset* field in the last entry of the FEA2 list must be zero.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code is the error code corresponding to the condition that caused the error. If no error occurred, *oError* is undefined.

DosSetPathInfo Parameter - cbInfoBuf

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

DosSetPathInfo Parameter - flOptions

flOptions (ULONG) - input
Information on how the set operation is to be performed.

If *flOptions* is 0x00000010 (DSPI_WRTTHRU), then all the information, including extended attributes (EAs), must be written to the disk before returning to the application. This guarantees that the EAs have been written to the disk. All other bits are reserved, and must be zero.

DosSetPathInfo Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetPathInfo returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
32	ERROR_SHARING_VIOLATION
87	ERROR_INVALID_PARAMETER
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE
122	ERROR_INSUFFICIENT_BUFFER
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosSetPathInfo - Parameters

pszPathName (PSZ) - input
Address of the ASCIIZ full path name of the file or subdirectory.

Global file-name characters are not permitted.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

ulInfoLevel (ULONG) - input
The level of file directory information being defined.

A value of 1 or 2 can be specified, as shown in the following list.

1	FIL_STANDARD Level 1 file information
2	FIL_QUERYEASIZE Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

pInfoBuf (PVOID) - input
Address of the storage area containing the file information being set.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure.

Level 2 sets a series of extended attribute (EA) name/value pairs.

Input	*pInfoBuf* contains an EAOP2 data structure. *fpGEA2List* is ignored. *fpFEA2List* points to a data area where the relevant FEA2 list is to be found. *oError* is ignored. The FEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the FEA2 list. The *oNextEntryOffset* field in the last entry of the FEA2 list must be zero.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code is the error code corresponding to the condition that caused the error. If no error occurred, *oError* is undefined.

cbInfoBuf (ULONG) - input
The length, in bytes, of *pInfoBuf*.

flOptions (ULONG) - input
Information on how the set operation is to be performed.

If *flOptions* is 0x00000010 (DSPI_WRTTHRU), then all the information, including extended attributes (EAs), must be written to the disk before returning to the application. This guarantees that the EAs have been written to the disk. All other bits are reserved, and must be zero.

ulrc (APIRET) - returns
Return Code.

DosSetPathInfo returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
32	ERROR_SHARING_VIOLATION
87	ERROR_INVALID_PARAMETER
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE
122	ERROR_INSUFFICIENT_BUFFER
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

For a full list of error codes, see Errors.

DosSetPathInfo - Remarks

To use DosSetPathInfo to set any level of file information for a file or subdirectory, a process must have exclusive write access to the closed file object. Thus, if the file object is already accessed by another process, any call to DosSetPathInfo will fail.

A value of 0 in the date and time components of a field causes that field to be left unchanged. For example, if both "last write date" and "last write time" are specified as 0 in the Level 1 information structure, then both attributes of the file are left unchanged. If either "last write date" or "last write time" are other than 0, then both attributes of the file are set to the new values.

For data integrity purposes, the Write-Through bit in *flOptions* should be used only to write the extended attributes to the disk immediately, instead of caching them and writing them later. Having the Write-Through bit set constantly can degrade performance.

In the FAT file system, only the dates and times of the last write can be modified. Creation and last-access dates and times are not affected.

The last-modification date and time will be changed if the extended attributes are modified.

DosSetPathInfo - Related Functions

Related Functions

- DosEnumAttribute
- DosQueryFileInfo
- DosQueryPathInfo
- DosQuerySysInfo
- DosSetFileInfo

DosSetPathInfo - Example Code

This example creates a directory named "HIDEME", makes it hidden, and finally deletes it.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {
   UCHAR       achNewDir[256]   = "\\HIDEME";           /* Directory name    */
   FILESTATUS3 fsts3PathInfo    = {{0}};                /* Directory info    */
   ULONG       ulBufferSize     = sizeof(FILESTATUS3);  /* Buffer size       */
   APIRET      rc               = NO_ERROR;             /* Return code       */

   rc = DosCreateDir(achNewDir, (PEAOP2) NULL);        /* Create directory
                                                          with no EAs       */
   if (rc != NO_ERROR) {
      printf("DosCreateDir error: return code = %u\n", rc);
      return 1;
   } else {
      printf("Directory %s created.\n",achNewDir);
   }

   rc = DosQueryPathInfo(achNewDir, FIL_STANDARD,
                         &fsts3PathInfo, ulBufferSize); /* Get standard info */
   if (rc != NO_ERROR) {
      printf("DosQueryPathInfo error: return code = %u\n", rc);
      return 1;
   }

   fsts3PathInfo.attrFile  = FILE_HIDDEN;   /* Add HIDDEN attribute to path */

   rc = DosSetPathInfo(achNewDir,           /* Change directory info on     */
                       FIL_STANDARD,        /* the disk using the buffer    */
                       &fsts3PathInfo,      /* we just updated.             */
                       ulBufferSize,
                       DSPI_WRTTHRU );      /* Write data before returning  */
   if (rc != NO_ERROR) {
       printf("DosSetPathInfo error: return code = %u\n", rc);
       return 1;
   } else {
       printf("Directory %s hidden.\n",achNewDir);
   }
       /* Delete the hidden directory.  If this step is omitted, the directory
          can still be manipulated by standard OS/2 commands like CHDIR and
          RMDIR, it will just not be displayed in a DIR command without the
          /AH display option specified.                                      */

   rc = DosDeleteDir (achNewDir);
```

```
   if (rc != NO_ERROR) {
       printf ("DosDeleteDir error : return code = %u\n", rc);
       return 1;
   } else {
       printf("Directory %s deleted.\n",achNewDir);
   }

   return NO_ERROR;
}
```

DosSetPathInfo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetPriority

DosSetPriority - Syntax

Changes the base priority of a child process or thread in the current process.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     scope;    /*  The extent of the priority change. */
ULONG     ulClass;  /*  Priority class of a process. */
LONG      delta;    /*  Change to apply to the current base priority level of the process. */
ULONG     PorTid;   /*  A process identifier (scope == PRTYS_PROCESS or PRTYS_PROCESSTREE) or a thread identifie
APIRET    ulrc;     /*  Return Code. */

ulrc = DosSetPriority(scope, ulClass, delta,
         PorTid);
```

DosSetPriority Parameter - scope

scope (ULONG) - input
The extent of the priority change.

The values of this field are shown in the following list:

0 PRTYS_PROCESS
All the threads of any process.

1 PRTYS_PROCESSTREE
All the threads of a process and any descendants. The indicated process must be the current process or a process created by the current process. Detached processes may not be specified. The indicated process may have terminated.

2 PRTYS_THREAD
A single thread of the current process.

DosSetPriority Parameter - ulClass

ulClass (ULONG) - input
Priority class of a process.

The values of this field are shown in the following list:

0 PRTYC_NOCHANGE
No change, leave as is

1 PRTYC_IDLETIME
Idle-time

2 PRTYC_REGULAR
Regular

3 PRTYC_TIMECRITICAL
Time-critical

4 PRTYC_FOREGROUNDSERVER
Fixed high

DosSetPriority Parameter - delta

delta (LONG) - input
Change to apply to the current base priority level of the process.

This value must range from -31 (PRTYD_MINIMUM) to +31 (PRTYD_MAXIMUM).

DosSetPriority Parameter - PorTid

PorTid (ULONG) - input
A process identifier (*scope* == PRTYS_PROCESS or PRTYS_PROCESSTREE) or a thread identifier (*scope* == PRTYS_THREAD).

If this operand is equal to zero, the current process or thread is assumed.

DosSetPriority Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetPriority returns one of the following values:

0	NO_ERROR
303	ERROR_INVALID_PROCID
304	ERROR_INVALID_PDELTA
305	ERROR_NOT_DESCENDANT
307	ERROR_INVALID_PCLASS
308	ERROR_INVALID_SCOPE
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosSetPriority - Parameters

scope (ULONG) - input
The extent of the priority change.

The values of this field are shown in the following list:

0 PRTYS_PROCESS
All the threads of any process.

1 PRTYS_PROCESSTREE
All the threads of a process and any descendants. The indicated process must be the current process or a process created by the current process. Detached processes may not be specified. The indicated process may have terminated.

2 PRTYS_THREAD
A single thread of the current process.

ulClass (ULONG) - input
Priority class of a process.

The values of this field are shown in the following list:

0 PRTYC_NOCHANGE
No change, leave as is

1 PRTYC_IDLETIME
Idle-time

2 PRTYC_REGULAR
Regular

3 PRTYC_TIMECRITICAL
Time-critical

4 PRTYC_FOREGROUNDSERVER
Fixed high

delta (LONG) - input
Change to apply to the current base priority level of the process.

This value must range from -31 (PRTYD_MINIMUM) to +31 (PRTYD_MAXIMUM).

PorTid (ULONG) - input
A process identifier (*scope* == PRTYS_PROCESS or PRTYS_PROCESSTREE) or a thread identifier (*scope* == PRTYS_THREAD).

If this operand is equal to zero, the current process or thread is assumed.

ulrc (APIRET) - returns
Return Code.

DosSetPriority returns one of the following values:

0	NO_ERROR

303	ERROR_INVALID_PROCID
304	ERROR_INVALID_PDELTA
305	ERROR_NOT_DESCENDANT
307	ERROR_INVALID_PCLASS
308	ERROR_INVALID_SCOPE
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosSetPriority - Remarks

DosSetPriority allows a process to change the priority of the current process or a child process, as well as any descendants. It also allows a process to change the priority of a single thread within the current process.

When a process changes the priority of current or descendant processes, only default priorities are changed.

DosSetPriority - Related Functions

Related Functions

- DosEnterCritSec
- DosGetInfoBlocks

DosSetPriority - Example Code

This sample sets the current thread priority to Time Critical level 15. It then uses DosGetInfoBlocks to retrieve the priority.

```
#define INCL_DOSPROCESS
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

int main(VOID)
{
   PTIB   ptib = NULL;          /* Thread information block structure  */
   PPIB   ppib = NULL;          /* Process information block structure */
   APIRET rc   = NO_ERROR;      /* Return code                         */

   rc = DosSetPriority (PRTYS_THREAD,        /* Change a single thread */
                        PRTYC_TIMECRITICAL,  /* Time critical class    */
                        15L,                 /* Increase by 15         */
                        0L);                 /* Assume current thread  */
   if (rc != NO_ERROR) {
      printf ("DosSetPriority error : rc = %u\n", rc);
      return 1;
   } else {
      rc = DosGetInfoBlocks(&ptib, &ppib);

      if (rc != NO_ERROR) {
         printf ("DosGetInfoBlocks error : rc = %u\n", rc);
         return 1;
      } else {
         printf("Priority Class = %d\n",
                ((ptib->tib_ptib2->tib2_ulpri) >> 8) & 0x00FF);
         printf("Priority Level = %d\n",
                ((ptib->tib_ptib2->tib2_ulpri) & 0x001F) );
      } /* endif */
   }

   return NO_ERROR;
}
```

DosSetPriority - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetProcessCp

DosSetProcessCp - Syntax

Allows a process to set its code page.

```
#define INCL_DOSNLS
#include <os2.h>

ULONG     cp;     /*  A code page identifier. */
APIRET    ulrc;   /*  Return Code. */

ulrc = DosSetProcessCp(cp);
```

DosSetProcessCp Parameter - cp

cp (ULONG) - input
A code page identifier.

Possible values are shown in the list below:

437	United States
850	Multilingual
852	Latin 2 (Czechoslovakia, Hungary, Poland, Yugoslavia)
857	Turkish
860	Portuguese
861	Iceland
863	Canadian French
865	Nordic
932	Japan
934	Korea
936	People's Republic of China
938	Taiwan
942	Japan SAA
944	Korea SAA

946	People's Republic of China SAA
948	Taiwan SAA

Note: Code pages 932, 934, 936, 938, 942, 944, 946, and 948 are supported only with the Asian version of the operating system on Asian hardware.

DosSetProcessCp Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetProcessCp returns one of the following values:

0	NO_ERROR
472	ERROR_INVALID_CODE_PAGE

For a full list of error codes, see Errors.

DosSetProcessCp - Parameters

cp (ULONG) - input
A code page identifier.

Possible values are shown in the list below:

437	United States
850	Multilingual
852	Latin 2 (Czechoslovakia, Hungary, Poland, Yugoslavia)
857	Turkish
860	Portuguese
861	Iceland
863	Canadian French
865	Nordic
932	Japan
934	Korea
936	People's Republic of China
938	Taiwan
942	Japan SAA
944	Korea SAA
946	People's Republic of China SAA
948	Taiwan SAA

Note: Code pages 932, 934, 936, 938, 942, 944, 946, and 948 are supported only with the Asian version of the operating system on Asian hardware.

ulrc (APIRET) - returns
Return Code.

DosSetProcessCp returns one of the following values:

0	NO_ERROR
472	ERROR_INVALID_CODE_PAGE

For a full list of error codes, see Errors.

DosSetProcessCp - Remarks

DosSetProcessCp sets the process code page of the calling process. The code page of a process is used in the following ways:

First, the printer code page is set to the process code page through the file system and printer spooler (the system spooler must be installed) when the process makes a request to open the printer. Calling DosSetProcessCp does not affect the code page of a printer opened prior to the call, and does not affect the code page of a printer opened by another process.

Second, country-dependent information, by default, is retrieved encoded in the code page of the calling process.

Third, a newly-created process inherits its process code page from its parent process.

DosSetProcessCp does not affect the display or keyboard code page.

DosSetProcessCp - Related Functions

Related Functions

- DosMapCase
- DosQueryCollate
- DosQueryCp
- DosQueryCtryInfo
- DosQueryDBCSEnv

DosSetProcessCp - Example Code

This example sets the code page of the process to 850.

```
#define INCL_DOSNLS       /* National Language Support values */
#define INCL_DOSERRORS    /* DOS Error values */
#include <os2.h>
#include <stdio.h>


int main(VOID) {
APIRET rc      = NO_ERROR;

rc = DosSetProcessCp(850);

if (rc != NO_ERROR) {
  printf("DosSetProcessCp error: return code = %u\n", rc);
  return 1;
}

return NO_ERROR;
}
```

DosSetProcessCp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetRelMaxFH

DosSetRelMaxFH - Syntax

Adjusts the maximum number of file handles for the calling process.

```
#define INCL_DOSFILEMGR
#include <os2.h>

PLONG     pcbReqCount;  /*  Address of the number to be added to the maximum number of file handles for the call
PULONG    pcbCurMaxFH;  /*  Address of the variable to receive the new total number of allocated file handles. *
APIRET    ulrc;         /*  Return Code. */

ulrc = DosSetRelMaxFH(pcbReqCount, pcbCurMaxFH);
```

DosSetRelMaxFH Parameter - pcbReqCount

pcbReqCount (PLONG) - input
Address of the number to be added to the maximum number of file handles for the calling process.

If *pcbReqCount* is positive, the maximum number of file handles is increased. If *pcbReqCount* is negative, the maximum number of file handles is decreased.

The system treats a decrease in the maximum number of file handles as an advisory request that may or may not be granted; the system may track and defer such a request.

DosSetRelMaxFH Parameter - pcbCurMaxFH

pcbCurMaxFH (PULONG) - output
Address of the variable to receive the new total number of allocated file handles.

DosSetRelMaxFH Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetRelMaxFH returns one of the following values:

0	NO_ERROR

For a full list of error codes, see Errors.

DosSetRelMaxFH - Parameters

pcbReqCount (PLONG) - input
Address of the number to be added to the maximum number of file handles for the calling process.

If *pcbReqCount* is positive, the maximum number of file handles is increased. If *pcbReqCount* is negative, the maximum number of file handles is decreased.

The system treats a decrease in the maximum number of file handles as an advisory request that may or may not be granted; the system may track and defer such a request.

pcbCurMaxFH (PULONG) - output
Address of the variable to receive the new total number of allocated file handles.

ulrc (APIRET) - returns
Return Code.

DosSetRelMaxFH returns one of the following values:

0	NO_ERROR

For a full list of error codes, see Errors.

DosSetRelMaxFH - Remarks

All file handles that are currently open are preserved. The system may defer or disregard a request to decrease the maximum number of file handles for the current process. The return code is set to NO_ERROR even if the system defers or disregards a request for a decrease.

You should examine the value of *pcbCurMaxFH* to determine the result of DosSetRelMaxFH.

DosSetRelMaxFH - Related Functions

Related Functions

- DosDupHandle
- DosOpen
- DosSetMaxFH

DosSetRelMaxFH - Example Code

This example shows how to define and adjust the maximum number of file handles.

```
#define INCL_DOSFILEMGR      /* File Manager values */
#define INCL_DOSERRORS       /* DOS Error values */
#include <os2.h>
#include <stdio.h>

int main(VOID){

  ULONG    CurMaxFH      = 0;          /* Current count of handles         */
  LONG     ReqCount      = 0;          /* Number to adjust file handles    */
  APIRET   rc            = NO_ERROR;   /* Return code                      */
```

```
  rc = DosSetRelMaxFH(&ReqCount,     /* Using 0 here will return the       */
                      &CurMaxFH);    /* current number of file handles     */

  if (rc != NO_ERROR) {
     printf("DosSetRelMaxFH error: return code = %u\n", rc);
     return 1;
  } else {
     printf("Maximum number of file handles is %u.\n", CurMaxFH);
  }

  rc = DosSetMaxFH(110L);            /* Set maximum file handles to 110  */

  if (rc != NO_ERROR) {
     printf("DosSetMaxFH error: return code = %u\n", rc);
     return 1;
  }

  ReqCount = -5L;                    /* Want 5 less file handles         */

  rc = DosSetRelMaxFH(&ReqCount,&CurMaxFH);     /* Change handle maximum */

  if (rc != NO_ERROR) {
     printf("DosSetRelMaxFH error: return code = %u\n", rc);
     return 1;
  } else {
     printf("Maximum number of file handles is now %u.\n", CurMaxFH);
  }

  return NO_ERROR;
}
```

DosSetRelMaxFH - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetSession

DosSetSession - Syntax

Sets the status of a child session.

```
#define INCL_dossesmgr
#include <os2.h>

ULONG          idSession;  /*  The identifier of the target session. */
PSTATUSDATA    psd;        /*  A pointer to the STATUSDATA which contains the session status data. */
APIRET         ulrc;       /*  Return code. */

ulrc = DosSetSession(idSession, psd);
```

DosSetSession Parameter - idSession

idSession (ULONG) - input
The identifier of the target session.

The value specified must have been returned on a previous call to DosStartSession.

DosSetSession Parameter - psd

psd (PSTATUSDATA) - input
A pointer to the STATUSDATA which contains the session status data.

DosSetSession Return Value - ulrc

ulrc (APIRET) - returns
Return code.

DosSetSession returns one of the following values:

0	NO_ERROR
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
455	ERROR_SMG_INVALID_BOND_OPTION
456	ERROR_SMG_INVALID_SELECT_OPT
460	ERROR_SMG_PROCESS_NOT_PARENT
461	ERROR_SMG_INVALID_DATA_LENGTH
463	ERROR_SMG_RETRY_SUB_ALLOC

For a full list of error codes, see Errors.

DosSetSession - Parameters

idSession (ULONG) - input
The identifier of the target session.

The value specified must have been returned on a previous call to DosStartSession.

psd (PSTATUSDATA) - input
A pointer to the STATUSDATA which contains the session status data.

ulrc (APIRET) - returns
Return code.

DosSetSession returns one of the following values:

0	NO_ERROR

369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
455	ERROR_SMG_INVALID_BOND_OPTION
456	ERROR_SMG_INVALID_SELECT_OPT
460	ERROR_SMG_PROCESS_NOT_PARENT
461	ERROR_SMG_INVALID_DATA_LENGTH
463	ERROR_SMG_RETRY_SUB_ALLOC

For a full list of error codes, see Errors.

DosSetSession - Remarks

DosSetSession sets or resets one or both of the following parameters related to a child session:

1. Selectable or nonselectable. This parameter allows a parent session to set one of its child sessions as selectable or nonselectable from the Shell switch list.

2. Bond/no bond. This parameter allows a parent session to bond one of its child sessions to itself. This means that if the operator subsequently selects the parent session from the Shell menu (or double-clicks to the parent session), then the child session will be brought to the foreground.

The parameters only affect user selections from the Shell switch list or Shell selections during system hot key processing. They do not affect selections made by the parent session. Thus, when a parent session selects its own session, its own session is brought to the foreground, even if a bond is in effect. When a parent session selects a child session, the child session is brought to the foreground, even if the parent has previously set the child nonselectable.

The above parameters may be set individually. Either can be changed without affecting the current setting of the other.

DosSetSession may only be issued by a parent session for a child session. Neither the parent session itself nor any grandchild, nor any other descendant session beyond a child session, may be the target of this function. DosSetSession may only be issued by the process that originally started the specified session (*idSession*) through DosStartSession.

DosSetSession may only be used to change the status of child sessions that were originally started by the caller with DosStartSession specifying a value of 1 for *Related*. That is, DosSetSession may not be used to change the status of sessions started as independent sessions.

A bond established between a parent session and a child session can be broken by reissuing DosSetSession and specifying either:

- *BondInd* = 2 to break the bond, or
- *BondInd* = 1 to establish a bond with a different child session. In this case, the bond with the previous child is broken.

If a bond is established between session A and its immediate child session B, and if another bond is established between session B and its immediate child session C, then if the operator selects session A, session C is brought to the foreground. However, if session A selects itself, session A is brought to the foreground. If session A selects session B, session C is brought to the foreground. In the latter case, the bond between B and C is honored.

Assume that a bond is established between session A and its immediate child session B, and assume that session B is nonselectable. The operator will not be able to select session B directly. However, if the operator selects session A, session B will be brought to the foreground.

A parent session may be running in either the foreground or the background when DosSetSession is issued.

DosSetSession - Related Functions

Related Functions

- DosSelectSession
- DosStartSession
- DosStopSession

DosSetSession - Example Code

This example removes a child session from a window list.

```
#define INCL_DOSPROCESS   /* DOS process values - needed for DosSleep */
#define INCL_DOSSESMGR
#define INCL_DOSERRORS
#include <stdio.h>
#include <os2.h>

int main(VOID) {
   STARTDATA SData       = {0};
   PSZ       PgmTitle    = "Not in the Window List",    /* Title      */
             PgmName     = "CMD.EXE";  /* This starts an OS/2 session */
   APIRET    rc          = NO_ERROR;   /* Return code                 */
   PID       pid         = 0;          /* PID returned                */
   ULONG     ulSessID    = 0;          /* Session ID returned         */
   UCHAR     achObjBuf[100] = {0};     /* Error info if start fails   */
   STATUSDATA ChildStatus   = {0};     /* Child status data           */

   SData.Length  = sizeof(STARTDATA);
   SData.Related = SSF_RELATED_CHILD;       /* A dependent session          */
   SData.FgBg    = SSF_FGBG_FORE;           /* start session in foreground  */
   SData.TraceOpt = SSF_TRACEOPT_NONE;      /* No trace                     */
            /* Start an OS/2 session using "CMD.EXE /K" */
   SData.PgmTitle = PgmTitle;
   SData.PgmName = PgmName;
   SData.PgmInputs = "/K";                     /* Keep session up           */
   SData.TermQ = 0;                            /* No termination queue      */
   SData.Environment = 0;                      /* No environment string     */
   SData.InheritOpt = SSF_INHERTOPT_SHELL;     /* Inherit shell's environ.  */
   SData.SessionType = SSF_TYPE_WINDOWABLEVIO; /* Windowed VIO session      */
   SData.IconFile = 0;                         /* No icon association       */
   SData.PgmHandle = 0;
   SData.PgmControl = SSF_CONTROL_VISIBLE | SSF_CONTROL_MAXIMIZE;
   SData.InitXPos  = 30;     /* Initial window coordinates              */
   SData.InitYPos  = 40;
   SData.InitXSize = 200;    /* Initial window size */
   SData.InitYSize = 140;
   SData.Reserved = 0;
   SData.ObjectBuffer  = achObjBuf; /* Contains info if DosExecPgm fails */
   SData.ObjectBuffLen = (ULONG) sizeof(achObjBuf);
   rc = DosStartSession(&SData, &ulSessID, &pid);  /* Start the session */
   if (rc != NO_ERROR) {
      printf ("DosStartSession error : return code = %u\n", rc);
      return 1;
   }
   printf("Removing child process from the Window List... \n");

   ChildStatus.Length = sizeof(STATUSDATA);
   ChildStatus.SelectInd = SET_SESSION_NON_SELECTABLE;
   ChildStatus.BondInd = SET_SESSION_UNCHANGED;

   rc = DosSetSession(ulSessID, &ChildStatus);
   if (rc != NO_ERROR) {
      printf ("DosSetSession error : return code = %u\n", rc);
      return 1;
   }

   printf("\nPress Ctrl-Esc  The child is no longer listed.\n");
   printf("\nProgram will terminate in 10 seconds...\n");
   rc = DosSleep(10000L);  /* wait 10 seconds */

   return NO_ERROR;
}
```

DosSetSession - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code

Related Functions
Glossary

DosSetSignalExceptionFocus

DosSetSignalExceptionFocus - Syntax

Causes the current process to become the focus for the Ctrl+C and Ctrl+Break signals.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

BOOL32    flag;      /*  Focus flag. */
PULONG    pulTimes;  /*  The number of times DosSetSignalExceptionFocus has been called by the current process w
APIRET    ulrc;      /*  Return Code. */

ulrc = DosSetSignalExceptionFocus(flag, pulTimes);
```

DosSetSignalExceptionFocus Parameter - flag

flag (BOOL32) - input
Focus flag.

This parameter may have either of two values:

0	SIG_UNSETFOCUS Stop receiving signals.
1	SIG_SETFOCUS Start receiving signals.

DosSetSignalExceptionFocus Parameter - pulTimes

pulTimes (PULONG) - output
The number of times DosSetSignalExceptionFocus has been called by the current process with *flag* set to SIG_SETFOCUS, minus the number of times it has been called with *flag* set to SIG_UNSETFOCUS.

DosSetSignalExceptionFocus Return Value - ulrc

ulrc (APIRET) - returns

Return Code.

DosSetSignalExceptionFocus returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
300	ERROR_ALREADY_RESET
303	ERROR_INVALID_PROCID
650	ERROR_NESTING_TOO_DEEP

For a full list of error codes, see Errors.

DosSetSignalExceptionFocus - Parameters

flag (BOOL32) - input
Focus flag.

This parameter may have either of two values:

0	SIG_UNSETFOCUS Stop receiving signals.
1	SIG_SETFOCUS Start receiving signals.

pulTimes (PULONG) - output
The number of times DosSetSignalExceptionFocus has been called by the current process with *flag* set to SIG_SETFOCUS, minus the number of times it has been called with *flag* set to SIG_UNSETFOCUS.

ulrc (APIRET) - returns
Return Code.

DosSetSignalExceptionFocus returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
300	ERROR_ALREADY_RESET
303	ERROR_INVALID_PROCID
650	ERROR_NESTING_TOO_DEEP

For a full list of error codes, see Errors.

DosSetSignalExceptionFocus - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosSetSignalExceptionFocus causes the calling process to become the signal focus for its screen group for the XCPT_SIGNAL_BREAK (Ctrl+Break) and XCPT_SIGNAL_INTR (Ctrl+C) signal exceptions.

You cannot issue DosSetSignalExceptionFocus from a Presentation Manager (PM) application. If you do, you get the return code ERROR_INVALID_PROCID. You can issue this function from a full-screen or windowed application.

For a detailed list of the system exceptions, see System Exceptions.

DosSetSignalExceptionFocus - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosExitMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetExceptionHandler
- DosUnsetExceptionHandler
- DosUnwindException

DosSetSignalExceptionFocus - Example Code

This example shows how to set the current thread to be the focus for the Ctrl+C and Ctrl+Break signals.

```
#define INCL_DOSPROCESS       /* DOS process values (for DosSleep) */
#define INCL_DOSEXCEPTIONS    /* DOS exception values */
#define INCL_ERRORS           /* DOS error values     */
#include <os2.h>
#include <stdio.h>

ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv );

int main (VOID)
{
  EXCEPTIONREGISTRATIONRECORD  RegRec  = {0}; /* Exception Registration Record */
  APIRET                       rc      = NO_ERROR;   /* Return code*/

  /* Add MyTermHandler to this thread's chain of exception handlers */

  RegRec.ExceptionHandler = (ERR)MyTermHandler;
  rc = DosSetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosSetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Terminate this program using Ctrl-C or Ctrl-Break.\n");

  rc = DosSleep(60000L);   /* Give user plenty of time to comply... */
  rc = DosUnsetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosUnsetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }
  return NO_ERROR;
}
/*****************************************************************************/
ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv )
{
  ULONG       ulCount = 0;          /* Count of API calls  */
  APIRET      rc      = NO_ERROR;   /* Return code         */

  printf("*** MyTermHandler entered: ExceptionNum = %x\n",p1->ExceptionNum);

   rc = DosSetSignalExceptionFocus( SIG_UNSETFOCUS, &ulCount);
   if (rc != NO_ERROR) {
      printf("DosSetSignalExceptionFocus error: return code = %u\n", rc);
      return 1;
   }
        /************************************/
        /* Handle XCPT_SIGNAL exceptions here */
        /************************************/

   rc = DosSetSignalExceptionFocus( SIG_SETFOCUS, &ulCount);
   if (rc != NO_ERROR) {
      printf("DosSetSignalExceptionFocus error: return code = %u\n", rc);
      return 1;
   }
```

```
  return XCPT_CONTINUE_SEARCH;          /* Exception not resolved... */
}
```

DosSetSignalExceptionFocus - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSetVerify

DosSetVerify - Syntax

Sets write verification.

```
#define INCL_DOSFILEMGR
#include <os2.h>

BOOL32    fVerifySetting;  /*  The state of verify mode. */
APIRET    ulrc;            /*  Return Code. */

ulrc = DosSetVerify(fVerifySetting);
```

DosSetVerify Parameter - fVerifySetting

fVerifySetting (BOOL32) - input
The state of verify mode.

Possible values are shown in the list below:

FALSE	Verify mode is not active.
TRUE	Verify mode is active.

DosSetVerify Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSetVerify returns one of the following values:

0	NO_ERROR
118	ERROR_INVALID_VERIFY_SWITCH

For a full list of error codes, see Errors.

DosSetVerify - Parameters

fVerifySetting (BOOL32) - input
The state of verify mode.

Possible values are shown in the list below:

FALSE	Verify mode is not active.
TRUE	Verify mode is active.

ulrc (APIRET) - returns
Return Code.

DosSetVerify returns one of the following values:

0	NO_ERROR
118	ERROR_INVALID_VERIFY_SWITCH

For a full list of error codes, see Errors.

DosSetVerify - Remarks

When verify mode is active, the operating system verifies that data written to the disk is recorded correctly, even though disk recording errors are rare.

DosSetVerify - Related Functions

Related Functions

- DosQueryVerify

DosSetVerify - Example Code

This example enables disk write verification. It creates a file named "VERIFY.DAT" and writes the critical data to it. After this is done, it restores the original settings.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
int main(VOID) {
   UCHAR  uchFileName[20]  = "VERIFY.DAT",     /* Name of file to use      */
          uchFileData[10] = "SampleData";      /* Stuff to put in file     */
   HFILE  hfFileHandle    = 0L;                /* Handle for user file     */
   BOOL32 fUserVerify     = 0;                 /* User Verify flag setting */
   ULONG  ulAction        = 0,                 /* Action done by DosOpen   */
          ulWritten       = 0;                 /* Number of bytes written  */
   APIRET rc              = NO_ERROR;          /* Return code              */

   rc = DosQueryVerify(&fUserVerify);     /* Get current setting of VERIFY */
   if (rc != NO_ERROR) {
     printf("DosQueryVerify error: return code = %u\n", rc);
     return 1;
   } else {
     printf ("Original setting of Verify=%s\n",(fUserVerify) ? "On" : "Off");
   } /* endif */

   rc = DosSetVerify(1);                  /* Set VERIFY=ON                */
   if (rc != NO_ERROR) {
     printf("DosSetVerify error: return code = %u\n", rc);
     return 1;  }

   /* Open the file VERIFY.DAT for read/write.  Create it if necessary */

   rc = DosOpen(uchFileName, &hfFileHandle, &ulAction,
                10L, FILE_NORMAL,
                OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_SHARE_DENYNONE  | OPEN_ACCESS_READWRITE, 0L);

   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1; }

         /* Write critical data to the file */

   rc = DosWrite (hfFileHandle, (PVOID) uchFileData,
                  sizeof(uchFileData), &ulWritten);
   if (rc == NO_ERROR) {
      printf ("%u bytes written to file %s with Verify=On\n",
               ulWritten,uchFileName);
   }

   rc = DosSetVerify(fUserVerify);         /* Restore user's verify value */
   if (rc != NO_ERROR) {
     printf("DosSetVerify error: return code = %u\n", rc);
     return 1;
   }
   rc = DosClose(hfFileHandle);            /* Close the file              */

   return NO_ERROR;
}
```

DosSetVerify - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosShutdown

DosShutdown - Syntax

Locks out changes to all file systems, and writes system buffers to the disk in preparation for turning off power to the system.

```
#define INCL_DOSFILEMGR
#include <os2.h>

ULONG     ulShutDownValue;  /*  Flag indicating the shutdown value. */
APIRET    ulrc;             /*  Return Code. */

ulrc = DosShutdown(ulShutDownValue);
```

DosShutdown Parameter - ulShutDownValue

ulShutDownValue (ULONG) - input
Flag indicating the shutdown value.

Possible values are described in the following list:

0
Perform full system shutdown and file-system lock.

1
Perform buffer and cache flushing to make system quiescent.

DosShutdown Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosShutdown returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
274	ERROR_ALREADY_SHUTDOWN

For a full list of error codes, see Errors.

DosShutdown - Parameters

ulShutDownValue (ULONG) - input
Flag indicating the shutdown value.

Possible values are described in the following list:

0
Perform full system shutdown and file-system lock.

1
Perform buffer and cache flushing to make system quiescent.

ulrc (APIRET) - returns
Return Code.

DosShutdown returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
274	ERROR_ALREADY_SHUTDOWN

For a full list of error codes, see Errors.

DosShutdown - Remarks

When *ulShutDownValue* equals 0, the full system is shut down and the file-system is locked. In this case, DosShutdown can take several minutes to complete its operation; the time depends on the amount of data in the buffers.

If other functions that change file-system data are issued while the system is shut down, either the return code ERROR_ALREADY_SHUTDOWN is set, or the other function calls are blocked permanently.

Allocated memory cannot be increased once DosShutdown has been issued. This means that in low-memory situations, some functions may fail because of a lack of memory. This is of particular importance to the process issuing DosShutdown. All memory that the calling process will ever need should be allocated before DosShutdown is issued. This includes implicit memory allocations that system functions make on behalf of DosShutdown.

When DosShutdown has completed successfully, the system can be powered-off or restarted.

When *ulShutDownValue* equals 1, the system is quiescent (no threads of execution are active) at the end of this function request, but the file-systems are not locked, and processing can resume at a later time.

DosShutdown - Example Code

This example prepares the file system for the shutdown of the system. Refer to the WinShutdownSystem function to perform a complete OS/2 shutdown.

```
#define INCL_DOSFILEMGR
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

int main(VOID)
  {

   APIRET  rc = NO_ERROR;    /* Return code */

   rc = DosShutdown(0L);

   if (rc != NO_ERROR) {
       printf("DosShutdown error: return code = %u\n",rc);
       return 1;
   } else {
       printf("The file system has been prepared for shutdown.\n");
   }  /* endif */

   return NO_ERROR;
   }
```

DosShutdown - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Glossary

DosSleep

DosSleep - Syntax

Suspends the current thread for a specified time interval.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     msec;  /*  The time, in milliseconds, for which the calling thread's execution will be suspended. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosSleep(msec);
```

DosSleep Parameter - msec

msec (ULONG) - input
The time, in milliseconds, for which the calling thread's execution will be suspended.

The system rounds this value up to the next clock tick.

DosSleep Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSleep returns one of the following values:

0	NO_ERROR
322	ERROR_TS_WAKEUP

For a full list of error codes, see Errors.

DosSleep - Parameters

msec (ULONG) - input
The time, in milliseconds, for which the calling thread's execution will be suspended.

The system rounds this value up to the next clock tick.

ulrc (APIRET) - returns
Return Code.

DosSleep returns one of the following values:

0	NO_ERROR
322	ERROR_TS_WAKEUP

For a full list of error codes, see Errors.

DosSleep - Remarks

DosSleep suspends the current thread for a specified time interval.

If a time interval of 0 is specified, the thread gives up the remainder of the current time slice, allowing any other ready threads of equal or higher priority to execute; the calling thread will execute again during its next scheduled time slice. If there is no other ready thread of equal or higher priority, DosSleep returns immediately; it does not give control to a thread of lower priority.

A time interval of 1 effectively causes the thread to give up the reminder of its current time slice and allow the next ready thread to execute, regardless of its priority. This technique gives lower priority threads a chance to execute, at least for the remainder of the time slice.

Time intervals for DosSleep, DosAsyncTimer, and DosStartTimer are specified in milliseconds; however, it is important to recognize that the actual duration of the specified time interval will be affected by two factors:

- First, the system clock keeps track of time in less precise units known as *clock ticks* . The duration of a clock tick depends on the frequency of the system-clock interrupt that is used by your computer. (To determine the duration of the clock tick on your computer, issue DosQuerySysInfo and examine the timer-interval field.)

 Because clock ticks are less precise than millisecond values, any time interval that is specified in milliseconds will be rounded up to the next clock tick.

- Second, because the system is a priority-based, multitasking operating system, there is no guarantee that a thread will resume immediately after the timer interval expires. If a higher-priority process or thread is executing, the timed thread blocks. (To minimize the inaccuracy caused by preemptive scheduling, an application can dedicate a thread to managing time-critical tasks and then raise that thread to a higher priority.)

In addition, the time interval for DosSleep refers to execution time (accumulated scheduled time slices), not to elapsed real time. Elapsed real time will be longer and will vary, depending on the hardware and on the number and priorities of other threads executing in the system. (Elapsed real time for the asynchronous timers, started by DosAsyncTimer and DosStartTimer, will be much closer to their specified time intervals because these timers run independently of the calling thread's execution.)

Because the above factors usually cause the sleep interval to be longer than requested (though generally within a few clock ticks), DosSleep should not be used as a substitute for a real-time clock.

To ensure optimal performance, do not use DosSleep in a single-thread Presentation Manager application. (See WinStartTimer.)

If the calling thread is awakened before the time interval expires (by a system exception, for example), ERROR_TS_WAKEUP is returned.

DosSleep - Related Functions

Related Functions

- DosAsyncTimer
- DosGetDateTime
- DosSetDateTime
- DosStartTimer
- DosStopTimer

DosSleep - Example Code

This example uses DosStartSession to execute a CHKDSK on the current drive in an OS/2 window, and uses DosSleep to wait 30 seconds before terminating.

```
#define INCL_DOSPROCESS
#define INCL_DOSSESMGR
#define INCL_DOSERRORS
#include <stdio.h>
#include <os2.h>

int main(VOID) {
   STARTDATA SData   = {0};
   PSZ       PgmTitle    = "CHKDSK for Current Drive",  /* Title      */
             PgmName     = "CHKDSK.COM";  /* Want to run CHKDSK       */
   APIRET    rc          = NO_ERROR;   /* Return code                 */
   PID       pidChild    = 0;          /* PID returned                */
   ULONG     ulSessID    = 0;          /* Session ID returned         */
   UCHAR     achObjBuf[100] = {0};     /* Buffer for error info       */

   SData.Length  = sizeof(STARTDATA);
   SData.Related = SSF_RELATED_CHILD;       /* Start a child session        */
   SData.FgBg    = SSF_FGBG_FORE;           /* Start session in foreground  */
   SData.TraceOpt = SSF_TRACEOPT_NONE;      /* No trace                     */

   SData.PgmTitle = PgmTitle;
   SData.PgmName = PgmName;
   SData.PgmInputs = ""; /* No parameters                                   */

   SData.InheritOpt = SSF_INHERTOPT_SHELL;     /* Inherit shell's environ.  */
   SData.SessionType = SSF_TYPE_WINDOWABLEVIO; /* Windowed VIO session      */
   SData.PgmControl = SSF_CONTROL_VISIBLE | SSF_CONTROL_NOAUTOCLOSE;
   SData.InitXPos  = 30;     /* Initial window coordinates                 */
   SData.InitYPos  = 40;
   SData.InitXSize = 200;    /* Initial window size                        */
   SData.InitYSize = 140;
   SData.ObjectBuffer  = achObjBuf; /* Contains info if DosExecPgm fails    */
   SData.ObjectBuffLen = (ULONG) sizeof(achObjBuf);

   rc = DosStartSession(&SData, &ulSessID, &pidChild);  /* Start CHKDSK     */
   if (rc != NO_ERROR) {
      printf ("DosStartSession error : return code = %u\n", rc);
      return 1;
   } else {
      printf ("Child process has SessID of %u and PID of %u.\n", ulSessID,pidChild);
   }

   printf ("Waiting 30 seconds before terminating...\n");

   rc = DosSleep(30000L);         /* Wait 30 seconds for child to complete */
   if (rc != NO_ERROR) {
      printf ("DosSleep error : return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
  }
```

DosSleep - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosStartSession

DosStartSession - Syntax

Allows an application to start another session, and to specify the name of the program to be started in that session.

```
#define INCL_dossesmgr
#include <os2.h>

PSTARTDATA    psd;         /*  A pointer to the STARTDATA structure containing data needed to start a window ses
PULONG        pidSession;  /*  A pointer to a ULONG in which the session identifier associated with the child se
PPID          ppid;        /*  A pointer to the PID in which the process identifier associated with the child pr
APIRET        ulrc;        /*  Return code. */

ulrc = DosStartSession(psd, pidSession, ppid);
```

DosStartSession Parameter - psd

psd (PSTARTDATA) - input
A pointer to the STARTDATA structure containing data needed to start a window session.

DosStartSession Parameter - pidSession

pidSession (PULONG) - output
A pointer to a ULONG in which the session identifier associated with the child session created is returned.

pidSession is returned only when the value specified for *Related* is 1 (SSF_RELATED_CHILD). The *pidSession* returned can be specified on subsequent calls to DosSelectSession, DosSetSession, and DosStopSession.

This parameter is not returned for independent sessions. For more information about independent sessions, refer to the *Related* field in the STARTDATA.

DosStartSession Parameter - ppid

ppid (PPID) - output
A pointer to the PID in which the process identifier associated with the child process created is returned.

ppid is returned only when the value specified for *Related* is 1 (SSF_RELATED_CHILD). The *ppid* returned may not be used on any system functions (for example, DosSetPriority) that require a parent process/child process relationship.

This parameter is not returned for independent sessions. For more information about independent sessions, refer to the *Related* field in the STARTDATA structure.

Refer to **"Parent/Child Relationship"** in the **Remarks** section for more information about child processes.

DosStartSession Return Value - ulrc

ulrc (APIRET) - returns
Return code.

DosStartSession returns one of the following values:

0	NO_ERROR
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
457	ERROR_SMG_START_IN_BACKGROUND
460	ERROR_SMG_PROCESS_NOT_PARENT
463	ERROR_SMG_RETRY_SUB_ALLOC

For a full list of error codes, see Errors.

DosStartSession - Parameters

psd (PSTARTDATA) - input
A pointer to the STARTDATA structure containing data needed to start a window session.

pidSession (PULONG) - output
A pointer to a ULONG in which the session identifier associated with the child session created is returned.

pidSession is returned only when the value specified for *Related* is 1 (SSF_RELATED_CHILD). The *pidSession* returned can be specified on subsequent calls to DosSelectSession, DosSetSession, and DosStopSession.

This parameter is not returned for independent sessions. For more information about independent sessions, refer to the *Related* field in the STARTDATA.

ppid (PPID) - output
A pointer to the PID in which the process identifier associated with the child process created is returned.

ppid is returned only when the value specified for *Related* is 1 (SSF_RELATED_CHILD). The *ppid* returned may not be used on any system functions (for example, DosSetPriority) that require a parent process/child process relationship.

This parameter is not returned for independent sessions. For more information about independent sessions, refer to the *Related* field in the STARTDATA structure.

Refer to **"Parent/Child Relationship"** in the **Remarks** section for more information about child processes.

ulrc (APIRET) - returns
Return code.

DosStartSession returns one of the following values:

0	NO_ERROR
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
457	ERROR_SMG_START_IN_BACKGROUND
460	ERROR_SMG_PROCESS_NOT_PARENT
463	ERROR_SMG_RETRY_SUB_ALLOC

For a full list of error codes, see Errors.

DosStartSession - Remarks

DosStartSession allows an application to start another session, and to specify the name of the program to be started in that session.

A session can be thought of as a logical console, consisting of buffers for the screen, keyboard, and mouse.

New sessions may only be started in the foreground when the caller's session (or one of the caller's descendant sessions) is currently executing in the foreground. The foreground session for windowed applications is the session of the application that owns the window focus. The new session appears in the Shell switch list.

Any protect-mode application may start any other protect-mode application in a new session, regardless of the issuing program's session type.

You may use DosExecPgm to start a process that is of the same type as the starting process. Process types include Presentation Manager, text-windowed, and full-screen. You may not use DosExecPgm to start a process that is of a different type than the starting process.

You must use DosStartSession to start a new process from a process that is of a different type. For example, use DosStartSession to start a Presentation Manager process from a non-Presentation Manager process.

If you call DosStartSession from a detached process, the function returns ERROR_SMG_INVALID_CALL. This happens regardless of whether the session is to be started in the foreground or in the background. Detached processes are programs that:

- Run in the background
- Run asynchronously (independently) from their parents
- Do not use the keyboard, mouse, or screen (other than VioPopups)

Detached processes are started when:

- Programs are started from the command line using the DETACH command
- Programs are started with DosExecPgm using the EXEC_BACKGROUND flag

See DosExecPgm for more information on detached processes.

Foreground/Background Considerations

DosStartSession will only start a new session in the foreground if the program issuing DosStartSession or a descendent session is executing in the foreground session. Otherwise, DosStartSession will override the foreground request and start the new session in the background. A unique error is returned, ERROR_SMG_START_IN_BACKGROUND, indicating that the new session was started in the background. An active popup window may also prevent the new session from starting in the foreground. The foreground session for windowed applications is the session of the application that owns the window focus. Therefore, when a windowed session is started in the foreground, the new session will be given the window focus.

Parent/Child Relationship

When you specify SSF_RELATED_CHILD for Related, DosStartSession establishes a parent session/child session relationship. A parent process/child process relationship is not established. The parent process is the shell process just as if the operator had started the program from the shell menu. Therefore, the *ppid* returned by DosStartSession may not be used with any system functions (for example, DosSetPriority) that require a parent process/child process relationship.

Once a process has issued DosStartSession specifying SSF_RELATED_CHILD for *Related*, no other process within that session can issue related DosStartSession functions until all the dependent sessions have ended.

Debugger Considerations

Debuggers may want to debug all processes associated with an application, no matter how the process was started (by DosExecPgm or DosStartSession). A special trace option, *TraceOpt* value 2, has been provided for this purpose. When a value of 2 is specified for *TraceOpt*, the debugger must also supply the name of an existing queue, and a value of 1 for *Related*, on the DosStartSession function.

The Session Manager notifies the debugger whenever a new session is created through DosStartSession from the initial session started with a value of 2 for *TraceOpt*, or from any descendant session. The queue is posted regardless of how the new session is started (related, independent, with or without inheritance).

Note: If the first session started by the parent process (debugger) ends, any messages generated by descendent sessions of the first session will not get posted to the parent process (debugger). For example, session A starts session B. Session B, in turn, starts session C and D. When session B ends, the messages from sessions C and D will not get posted to session A.

Sessions started without inheritance are executed for tracing. It is the responsibility of the debugger to resume execution of the new process.

The debugger must issue DosReadQueue to receive notification when a child session is created. The word containing the request parameter, returned by DosReadQueue, will have a value of 1. The data element structure has the following format:

Size	Description
USHORT	Session ID
USHORT	Process ID

The debugger should issue DosReadQueue with the *wait* parameter set to 0. This is the only process that has addressability to the notification data element. After reading and processing the data element, the debugger must free the segment that contains the data element by issuing DosFreeMem.

The debugger may use DosSelectSession to switch itself or any descendant session into the foreground whenever the current foreground session is a descendant of the debugger.

PgmName and PgmInputs Considerations

The program identified by *PgmName* is executed directly, with no intermediate secondary command (CMD.EXE) process. Alternatively, the program can be executed indirectly through a secondary command (CMD.EXE) process by specifying CMD.EXE for *PgmName*, and by specifying either /C or /K followed by the file specification of the application to be loaded for *PgmInputs*. If the /C parameter is inserted at the beginning of the *PgmInputs* string, then when the application program ends, the session ends. If the /K parameter is inserted at the beginning of the *PgmInputs* string, then when the application ends, the operator sees the system command line prompt displayed. The operator can then either enter the name of another program or command to execute, or enter the EXIT command to end the session.

When the *PgmName* address is 0, or the ASCIIZ string is null, the program identified by the *PgmHandle* is started in the new session. If the *PgmHandle* is not specified, then the program specified as a parameter to the protect mode shell on the OS2_SHELL statement, or on the SHELL statement for a DOS session, in the configuration file (CONFIG.SYS) is executed and passed the specified *PgmInputs*. The default is the program name for the command processor (CMD.EXE for a non-DOS session, or COMMAND.COM for a DOS session).

The *PgmName* and *PgmInputs* strings combined length may not exceed 1024 characters.

Program Handle Considerations

If a process issues DosStartSession specifying only the program handle, then it must change to the working directory before issuing DosStartSession, and start the new process as inherited. If a process is started as non-inherited, it is up to that process to change to the correct directory.

Parent/Child Termination Considerations

The parent must create the termination queue prior to specifying the queue name on DosStartSession. The Session Manager will continue to notify the parent session through the specified queue as long as the process issuing DosStartSession remains a parent session. When all the child sessions for a particular parent session end, the termination queue is closed by the Session Manager. An existing queue name must be specified on the next DosStartSession function if the caller wants to continue receiving termination notification messages. The caller must use the same queue for all calls to DosStartSession.

The Session Manager writes a data element into the specified queue when any child session ends. The queue is posted regardless of who terminates the child session (for example, child, parent, or operator) and whether the termination is normal or abnormal.

A parent session issues DosReadQueue to receive notification when a child session has ended. The word that contains the request parameter, returned by DosReadQueue will be 0. The data element structure has the following format:

Size	Description
USHORT	Session ID of child
USHORT	Result code

The process that originally issued DosStartSession should issue DosReadQueue with the *wait* parameter set to 0. This is the only process that has addressability to the notification data element. After reading and processing the data element, the caller must free the segment containing the data element by issuing DosFreeMem.

An application may use the termination queue for additional interprocess communication, provided that a unique request identifier is passed via DosWriteQueue. Request identifier values 0 through 99 are reserved for the operating system. Request identifier values equal to or greater than 100 are available for application use.

When a child session ends, the result code returned in the *TermQ* data element is the result code of the program specified by *PgmName* assuming either:

- the program is executed directly, with no intermediate secondary command (CMD.EXE) process, or
- the program is executed indirectly through a secondary command (CMD.EXE) process, and the /C parameter is specified.

Otherwise, the result code of CMD.EXE is returned.

When a child session is executing in the foreground at the time it ends, the parent session becomes the foreground session. When a parent session ends, all child sessions that it created with DosStartSession, specifying a value of 1 for *Related*, are ended. When an independent session, created specifying a value of 0 for *Related*, ends in the foreground, the Shell selects the next foreground session.

Grandchildren Considerations

A session started through DosStartSession may issue DosStartSession. The following rules apply:

- The *pidSession* , specified on DosSelectSession, DosSetSession, and DosStopSession may only be the session identifier *(pidSession*) of an immediate child session, not a grandchild session or any descendant other than an immediate child session.
- If a bond is established between session A and its immediate child session B, and if another bond is established between session B and its immediate child session C, then if session A is selected, session C is brought to the foreground. Refer to DosSetSession for a description of what establishing a bond means.
- When a session ends, all of its descendants (child sessions, grandchild sessions, and so on) are ended.

DosStartSession - Related Functions

Related Functions

- DosSelectSession
- DosSetSession
- DosStopSession

DosStartSession - Example Code

This example starts a windowed OS/2 session.

```
#define INCL_DOSSESMGR
#define INCL_DOSERRORS
#include <stdio.h>
#include <os2.h>

int main(VOID) {
   STARTDATA SData       = {0};
   PSZ       PgmTitle    = "Test Program", /* Title of new session     */
             PgmName     = "CMD.EXE";  /* This starts an OS/2 session  */
   APIRET    rc          = NO_ERROR;   /* Return code                  */
   PID       pid         = 0;          /* PID returned                 */
   ULONG     ulSessID    = 0;          /* Session ID returned          */
   UCHAR     achObjBuf[256] = {0};     /* Error data if DosStart fails */

   SData.Length  = sizeof(STARTDATA);
   SData.Related = SSF_RELATED_INDEPENDENT; /* start an independent session */
   SData.FgBg    = SSF_FGBG_FORE;           /* start session in foreground  */
   SData.TraceOpt = SSF_TRACEOPT_NONE;      /* No trace                     */
            /* Start an OS/2 session using "CMD.EXE /K" */
   SData.PgmTitle = PgmTitle;
   SData.PgmName = PgmName;
   SData.PgmInputs = "/K";                     /* Keep session up           */

   SData.TermQ = 0;                            /* No termination queue      */
   SData.Environment = 0;                      /* No environment string     */
   SData.InheritOpt = SSF_INHERTOPT_SHELL;     /* Inherit shell's environ.  */
   SData.SessionType = SSF_TYPE_WINDOWABLEVIO; /* Windowed VIO session      */
   SData.IconFile = 0;                         /* No icon association       */
   SData.PgmHandle = 0;
           /* Open the session VISIBLE and MAXIMIZED */
   SData.PgmControl = SSF_CONTROL_VISIBLE | SSF_CONTROL_MAXIMIZE;
   SData.InitXPos  = 30;     /* Initial window coordinates              */
   SData.InitYPos  = 40;
   SData.InitXSize = 200;    /* Initial window size */
   SData.InitYSize = 140;
   SData.Reserved = 0;
   SData.ObjectBuffer  = achObjBuf; /* Contains info if DosExecPgm fails */
   SData.ObjectBuffLen = (ULONG) sizeof(achObjBuf);

   rc = DosStartSession(&SData, &ulSessID, &pid);  /* Start the session */

   if (rc != NO_ERROR) {
      printf ("DosStartSession error : return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
```

}

DosStartSession - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosStartTimer

DosStartTimer - Syntax

Starts an asynchronous, repeated-interval timer.

```
#define INCL_DOSDATETIME
#include <os2.h>

ULONG      msec;     /*  The time, in milliseconds, that will elapse between postings of the event semaphore spe
HSEM       hsem;     /*  The handle of the event semaphore that is posted each time msec elapses. */
PHTIMER    phtimer;  /*  A pointer to the timer handle. */
APIRET     ulrc;     /*  Return Code. */

ulrc = DosStartTimer(msec, hsem, phtimer);
```

DosStartTimer Parameter - msec

msec (ULONG) - input
The time, in milliseconds, that will elapse between postings of the event semaphore specified by *hsem* .

The system rounds this value up to the next clock tick.

DosStartTimer Parameter - hsem

hsem (HSEM) - input
The handle of the event semaphore that is posted each time *msec* elapses.

This semaphore must be a shared event semaphore. It should be reset between postings by calling DosResetEventSem.

DosStartTimer Parameter - phtimer

phtimer (PHTIMER) - output
A pointer to the timer handle.

This handle can be passed to DosStopTimer to stop the repeated-interval timer.

DosStartTimer Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosStartTimer returns one of the following values:

0	NO_ERROR
323	ERROR_TS_SEMHANDLE
324	ERROR_TS_NOTIMER

For a full list of error codes, see Errors.

DosStartTimer - Parameters

msec (ULONG) - input
The time, in milliseconds, that will elapse between postings of the event semaphore specified by *hsem* .

The system rounds this value up to the next clock tick.

hsem (HSEM) - input
The handle of the event semaphore that is posted each time *msec* elapses.

This semaphore must be a shared event semaphore. It should be reset between postings by calling DosResetEventSem.

phtimer (PHTIMER) - output
A pointer to the timer handle.

This handle can be passed to DosStopTimer to stop the repeated-interval timer.

ulrc (APIRET) - returns
Return Code.

DosStartTimer returns one of the following values:

0	NO_ERROR
323	ERROR_TS_SEMHANDLE
324	ERROR_TS_NOTIMER

For a full list of error codes, see Errors.

DosStartTimer - Remarks

DosStartTimer starts an asynchronous, repeated-interval timer, and posts an event semaphore each time the specified time interval expires.

Time intervals for DosStartTimer, DosAsyncTimer, and DosSleep are specified in milliseconds; however, it is important to recognize that the actual duration of the specified time interval will be affected by two factors:

- First, the system clock keeps track of time in less precise units known as *clock ticks* . The duration of a clock tick depends on the frequency of the system-clock interrupt that is used by your computer. (To determine the duration of the clock tick on your computer, issue DosQuerySysInfo and examine the *timer-interval* field.)

 Because clock ticks are less precise than millisecond values, any time interval that is specified in milliseconds will be rounded up to the next clock tick.

- Second, because the system is a priority-based, multitasking operating system, there is no guarantee that a thread will resume immediately after the timer interval expires. If a higher-priority process or thread is running, or if a hardware interrupt occurs, the timed thread blocks. (To minimize the inaccuracy caused by preemptive scheduling, an application can dedicate a thread to managing time-critical tasks and then raise that thread to a higher priority.)

These factors usually cause the timer interval to be longer than requested; however, it will generally be within a few clock ticks.

DosStartTimer - Related Functions

Related Functions

- DosAsyncTimer
- DosCreateEventSem
- DosGetDateTime
- DosOpenEventSem
- DosResetEventSem
- DosSetDateTime
- DosSleep
- DosStopTimer
- DosWaitEventSem

DosStartTimer - Example Code

This example sets up a periodic interval timer of 2 seconds, and lets it run for a while.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
APIRET  rc            = NO_ERROR;           /* Return code               */
ULONG   ulPostCount   = 0;                  /* Semaphore post count      */
ULONG   i             = 0;                  /* A loop index              */

   rc = DosCreateEventSem(NULL,           /* Unnamed semaphore            */
                          &hevEvent1,     /* Handle of semaphore returned */
                          DC_SEM_SHARED,  /* Indicate a shared semaphore  */
                          FALSE);         /* Put in RESET state           */
   if (rc != NO_ERROR) {
       printf("DosCreateEventSem error: return code = %u\n", rc);
       return 1;
   }
   rc = DosStartTimer(2000L,              /* 2 second interval            */
                      (HSEM) hevEvent1,   /* Semaphore to post            */
                      &htimerEvent1);     /* Timer handler (returned)     */
   if (rc != NO_ERROR) {
       printf("DosStartTimer error: return code = %u\n", rc);
       return 1;
```

```
    }

    for (i = 1 ; i < 6 ; i++) {

      rc = DosWaitEventSem(hevEvent1,15000L); /* Wait 15 seconds for timer */
      if (rc != NO_ERROR) {
          printf("DosWaitEventSem error: return code = %u\n", rc);
          return 1;
      }

      rc = DosResetEventSem(hevEvent1,       /* Reset the semaphore         */
                            &ulPostCount);   /* And get count (should be 1) */
      if (rc != NO_ERROR) {
          printf("DosWaitEventSem error: return code = %u\n", rc);
          return 1;
      }

      printf("Iteration %u: ulPostCount = %u\n", i, ulPostCount);

    } /* for loop */

    rc = DosStopTimer(htimerEvent1);       /* Stop the timer             */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u\n", rc);
        return 1;
    }

    rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u\n", rc);
        return 1;
    }

 return NO_ERROR;
}
```

DosStartTimer - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosStopSession

DosStopSession - Syntax

Ends one or all child sessions.

```
#define INCL_dossesmgr
#include <os2.h>

ULONG     scope;      /*  An indicator that specifies whether only the session specified by idSession, or all se
ULONG     idSession;  /*  The identifier of the session to be ended. */
```

```
APIRET    ulrc;       /*  Return code. */

ulrc = DosStopSession(scope, idSession);
```

DosStopSession Parameter - scope

scope (ULONG) - input
An indicator that specifies whether only the session specified by *idSession* , or all sessions, should be ended.

Possible values are shown in the following list:

0	STOP_SESSION_SPECIFIED Ends only the specified session.
1	STOP_SESSION_ALL Ends all sessions.

DosStopSession Parameter - idSession

idSession (ULONG) - input
The identifier of the session to be ended.

The value specified for *idSession* must have been returned on a previous call to DosStartSession. *idSession* is ignored if *scope* is equal to STOP_SESSION_ALL.

DosStopSession Return Value - ulrc

ulrc (APIRET) - returns
Return code.

DosStopSession returns one of the following values:

0	NO_ERROR
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
458	ERROR_SMG_INVALID_STOP_OPTION
459	ERROR_SMG_BAD_RESERVE
460	ERROR_SMG_PROCESS_NOT_PARENT
463	ERROR_SMG_RETRY_SUB_ALLOC

For a full list of error codes, see Errors.

DosStopSession - Parameters

scope (ULONG) - input
An indicator that specifies whether only the session specified by *idSession* , or all sessions, should be ended.

Possible values are shown in the following list:

0	STOP_SESSION_SPECIFIED Ends only the specified session.
1	STOP_SESSION_ALL Ends all sessions.

idSession (ULONG) - input
The identifier of the session to be ended.

The value specified for *idSession* must have been returned on a previous call to DosStartSession. *idSession* is ignored if *scope* is equal to STOP_SESSION_ALL.

ulrc (APIRET) - returns
Return code.

DosStopSession returns one of the following values:

0	NO_ERROR
369	ERROR_SMG_INVALID_SESSION_ID
418	ERROR_SMG_INVALID_CALL
458	ERROR_SMG_INVALID_STOP_OPTION
459	ERROR_SMG_BAD_RESERVE
460	ERROR_SMG_PROCESS_NOT_PARENT
463	ERROR_SMG_RETRY_SUB_ALLOC

For a full list of error codes, see Errors.

DosStopSession - Remarks

DosStopSession ends one or all child sessions.

DosStopSession may only be issued by a parent session for a child session. Neither the parent session itself nor any grandchild, nor any other descendant session beyond a child session, may be the target of this function. DosStopSession may only be issued by the process that originally started the specified session (*idSession*) with DosStartSession.

DosStopSession may only be used to end child sessions that were originally started by the caller with DosStartSession specifying SSF_RELATED_CHILD for *Related*. That is, sessions started as independent sessions may not be stopped.

If the child session specified with DosStopSession has related sessions, these sessions will also be ended.

If a child session is executing in the foreground at the time it is ended, the parent session becomes the foreground session. DosStopSession breaks any bond that existed between the parent session and the specified child session.

A parent session may be executing in either the foreground or background when DosStopSession is issued.

Since any process executing in the specified session may refuse to end, the only way to guarantee that the target session has ended is to wait for notification through the termination queue specified with DosStartSession.

DosStopSession - Related Functions

Related Functions

- DosSelectSession
- DosSetSession
- DosStartSession

DosStopSession - Example Code

This example starts and stops a child session.

```
#define INCL_DOSPROCESS   /* DOS process values - needed for DosSleep */
#define INCL_DOSSESMGR
#define INCL_DOSERRORS
#include <stdio.h>
#include <os2.h>

int main(VOID) {
   STARTDATA SData       = {0};
   PSZ       PgmTitle    = "This session will stop in a few moments...",
             PgmName     = "CMD.EXE";  /* This starts an OS/2 session */
   APIRET    rc          = NO_ERROR;   /* Return code                 */
   PID       pid         = 0;          /* PID returned                */
   ULONG     ulSessID    = 0;          /* Session ID returned         */
   UCHAR     achObjBuf[100] = {0};

   SData.Length  = sizeof(STARTDATA);
   SData.Related = SSF_RELATED_CHILD;       /* start a dependent session    */
   SData.FgBg    = SSF_FGBG_FORE;           /* start session in foreground  */
   SData.TraceOpt = SSF_TRACEOPT_NONE;      /* No trace                     */
            /* Start an OS/2 session using "CMD.EXE /K" */
   SData.PgmTitle = PgmTitle;
   SData.PgmName = PgmName;
   SData.PgmInputs = "/K";                     /* Keep session up           */
   SData.TermQ = 0;                            /* No termination queue      */
   SData.Environment = 0;                      /* No environment string     */
   SData.InheritOpt = SSF_INHERTOPT_SHELL;     /* Inherit shell's environ.  */
   SData.SessionType = SSF_TYPE_WINDOWABLEVIO; /* Windowed VIO session      */
   SData.IconFile = 0;                         /* No icon association       */
   SData.PgmHandle = 0;
   SData.PgmControl = SSF_CONTROL_VISIBLE | SSF_CONTROL_MAXIMIZE;
   SData.InitXPos  = 30;     /* Initial window coordinates              */
   SData.InitYPos  = 40;
   SData.InitXSize = 200;    /* Initial window size */
   SData.InitYSize = 140;
   SData.Reserved = 0;
   SData.ObjectBuffer  = achObjBuf; /* Contains info if DosExecPgm fails */
   SData.ObjectBuffLen = (ULONG) sizeof(achObjBuf);

   rc = DosStartSession(&SData, &ulSessID, &pid);  /* Start the session */
   if (rc != NO_ERROR) {
      printf ("DosStartSession error : return code = %u\n", rc);
      return 1;
   }

   printf("Waiting a few seconds... then child session will be stopped.\n");
   rc = DosSleep(5000L);

   rc = DosStopSession(STOP_SESSION_SPECIFIED, ulSessID);
   if (rc != NO_ERROR) {
      printf ("DosStopSession error : return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosStopSession - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosStopTimer

DosStopTimer - Syntax

Stops an asynchronous timer.

```
#define INCL_DOSDATETIME
#include <os2.h>

HTIMER    htimer;  /*  The handle of the timer to stop. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosStopTimer(htimer);
```

DosStopTimer Parameter - htimer

htimer (HTIMER) - input
The handle of the timer to stop.

DosStopTimer Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosStopTimer returns one of the following values:

0	NO_ERROR
326	ERROR_TS_HANDLE

For a full list of error codes, see Errors.

DosStopTimer - Parameters

htimer (HTIMER) - input
The handle of the timer to stop.

ulrc (APIRET) - returns
Return Code.

DosStopTimer returns one of the following values:

0	NO_ERROR
326	ERROR_TS_HANDLE

For a full list of error codes, see Errors.

DosStopTimer - Remarks

DosStopTimer stops either a repeated-interval timer (started by DosStartTimer), or a single-interval timer (started by DosAsyncTimer).

When DosStopTimer is called, no assumption can be made about the state of the event semaphore that is associated with the timer. If the application is going to reuse the semaphore in conjunction with another timer, it should issue DosResetEventSem to ensure that the semaphore is in the "reset" state before starting the timer.

DosStopTimer - Related Functions

Related Functions

- DosAsyncTimer
- DosGetDateTime
- DosSetDateTime
- DosSleep
- DosStartTimer

DosStopTimer - Example Code

This example sets up a periodic interval timer of 2 seconds, lets it run for a while, and finally stops it.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
APIRET  rc            = NO_ERROR;           /* Return code               */
ULONG   ulPostCount   = 0;                  /* Semaphore post count      */
ULONG   i             = 0;                  /* A loop index              */

   rc = DosCreateEventSem(NULL,           /* Unnamed semaphore            */
                          &hevEvent1,     /* Handle of semaphore returned */
                          DC_SEM_SHARED,  /* Indicate a shared semaphore  */
                          FALSE);         /* Put in RESET state           */
   if (rc != NO_ERROR) {
       printf("DosCreateEventSem error: return code = %u\n", rc);
       return 1;
   }

   rc = DosStartTimer(2000L,              /* 2 second interval            */
                      (HSEM) hevEvent1,   /* Semaphore to post            */
                      &htimerEvent1);     /* Timer handler (returned)     */
   if (rc != NO_ERROR) {
       printf("DosStartTimer error: return code = %u\n", rc);
       return 1;
   }

   for (i = 1 ; i < 6 ; i++) {

     rc = DosWaitEventSem(hevEvent1,15000L); /* Wait 15 seconds for timer */
     if (rc != NO_ERROR) {
         printf("DosWaitEventSem error: return code = %u\n", rc);
         return 1;
     }

     rc = DosResetEventSem(hevEvent1,       /* Reset the semaphore         */
                           &ulPostCount);   /* And get count (should be 1) */
```

```
      if (rc != NO_ERROR) {
          printf("DosWaitEventSem error: return code = %u\n", rc);
          return 1;
      }

      printf("Iteration %u: ulPostCount = %u\n", i, ulPostCount);

    } /* for loop */

    rc = DosStopTimer(htimerEvent1);       /* Stop the timer             */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u\n", rc);
        return 1;
    }
    rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
    if (rc != NO_ERROR) {
        printf("DosCloseEventSem error: return code = %u\n", rc);
        return 1;
    }

 return NO_ERROR;
}
```

DosStopTimer - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSubAllocMem

DosSubAllocMem - Syntax

Allocates a block of memory from a memory pool that was previously initialized by DosSubSetMem.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pbBase;  /*  The offset to the memory pool from which the block should be allocated. */
PPVOID    ppb;     /*  A pointer to a PVOID in which the offset of the allocated memory block is returned. */
ULONG     cb;      /*  The size, in bytes, of the memory block requested. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosSubAllocMem(pbBase, ppb, cb);
```

DosSubAllocMem Parameter - pbBase

pbBase (PVOID) - input
The offset to the memory pool from which the block should be allocated.

DosSubAllocMem Parameter - ppb

ppb (PPVOID) - output
A pointer to a PVOID in which the offset of the allocated memory block is returned.

DosSubAllocMem Parameter - cb

cb (ULONG) - input
The size, in bytes, of the memory block requested.

DosSubAllocMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSubAllocMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
311	ERROR_DOSSUB_NOMEM
532	ERROR_DOSSUB_CORRUPTED

For a full list of error codes, see Errors.

DosSubAllocMem - Parameters

pbBase (PVOID) - input
The offset to the memory pool from which the block should be allocated.

ppb (PPVOID) - output
A pointer to a PVOID in which the offset of the allocated memory block is returned.

cb (ULONG) - input
The size, in bytes, of the memory block requested.

ulrc (APIRET) - returns
Return Code.

DosSubAllocMem returns one of the following values:

0	NO_ERROR

87	ERROR_INVALID_PARAMETER
311	ERROR_DOSSUB_NOMEM
532	ERROR_DOSSUB_CORRUPTED

For a full list of error codes, see Errors.

DosSubAllocMem - Remarks

DosSubAllocMem allocates a block of memory from a memory pool previously initialized by DosSubSetMem.

Allocation size should be a multiple of 8 bytes, otherwise it will be rounded up. The maximum value for *cb* is the size of the memory pool initialized by DosSubSetMem minus 64 bytes.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosSubAllocMem - Related Functions

Related Functions

- DosSubFreeMem
- DosSubSetMem
- DosSubUnsetMem

DosSubAllocMem - Example Code

This example creates a heap to manage small amounts of memory. It finally frees the memory object.

```
#define INCL_DOSMEMMGR
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

#define NUMBLOCKS  128               /* Number of blocks in pool          */
#define POOLSIZE   (NUMBLOCKS*80+64) /* 128 blocks of 80 bytes each.  OS/2
                                        needs 64 bytes to manage the pool */
int main (VOID)
{
  PVOID   pvMemBase       = NULL;       /* Pointer to memory object (pool)*/
  PVOID   apvBlock[NUMBLOCKS] = {NULL}; /* Pointers to memory blocks      */
  ULONG   i               = 0;          /* A loop index                   */
  APIRET  rc              = NO_ERROR;   /* Return code                    */

    /* Allocate the memory pool.  Note the pool is intentionally NOT
       committed here.  OS/2 will commit the memory as needed.          */
```

```
  rc = DosAllocMem( &pvMemBase, POOLSIZE, PAG_WRITE );
  if (rc != NO_ERROR) {
    printf("DosAllocMem error:  return code = %u\n", rc);
    return 1;
  }
       /* Make the entire memory object available for Suballocation */

  rc = DosSubSetMem( pvMemBase, DOSSUB_INIT | DOSSUB_SPARSE_OBJ, POOLSIZE );
  if (rc != NO_ERROR) {
    printf("DosSubSetMem error:  return code = %u\n", rc);
    return 1;
  } else { printf("Memory object ready for suballocation.\n"); }

   /* Suballocate the pool into 128 memory blocks, each 80 bytes in size */

  for( i = 0; i < NUMBLOCKS; i++ ) {
    rc = DosSubAllocMem( pvMemBase, &apvBlock[ i ], 80 );
    if( rc != NO_ERROR ) {
      printf( "DosSubAllocMem error: return code = %u  (i = %u)\n", rc, i );
      break;             }
  }

                     /*    USE THE POOL HERE... */

              /* When done, free the 128 memory blocks */

  do {
    rc = DosSubFreeMem(  pvMemBase, apvBlock[ --i ], 80 );
    if (rc != NO_ERROR) {
      printf("DosSubFreeMem error:  return code = %u  (i = %u)\n", rc, i);
      return 1;
    }
  } while( i );
        /* Stop memory block management, and free the memory object */

  rc = DosSubUnsetMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosSubUnsetMem error:  return code = %u\n", rc);
    return 1;
  }

  rc = DosFreeMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosFreeMem error:  return code = %u\n", rc);
    return 1;
  } else {
    printf("Memory freed.\n");
  }

  return NO_ERROR;
}
```

DosSubAllocMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSubFreeMem

DosSubFreeMem - Syntax

Frees a block of memory that was previously allocated by DosSubAllocMem.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pbBase;  /*  The offset of the memory pool to which the block is to be freed. */
PVOID     pb;      /*  The offset of the memory block to be freed. */
ULONG     cb;      /*  The size, in bytes, of the memory block to be freed. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosSubFreeMem(pbBase, pb, cb);
```

DosSubFreeMem Parameter - pbBase

pbBase (PVOID) - input
The offset of the memory pool to which the block is to be freed.

DosSubFreeMem Parameter - pb

pb (PVOID) - input
The offset of the memory block to be freed.

The value specified must equal the *pb* value returned on a previous DosSubAllocMem function.

DosSubFreeMem Parameter - cb

cb (ULONG) - input
The size, in bytes, of the memory block to be freed.

DosSubFreeMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSubFreeMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
312	ERROR_DOSSUB_OVERLAP

532	ERROR_DOSSUB_CORRUPTED

For a full list of error codes, see Errors.

DosSubFreeMem - Parameters

pbBase (PVOID) - input
The offset of the memory pool to which the block is to be freed.

pb (PVOID) - input
The offset of the memory block to be freed.

The value specified must equal the *pb* value returned on a previous DosSubAllocMem function.

cb (ULONG) - input
The size, in bytes, of the memory block to be freed.

ulrc (APIRET) - returns
Return Code.

DosSubFreeMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
312	ERROR_DOSSUB_OVERLAP
532	ERROR_DOSSUB_CORRUPTED

For a full list of error codes, see Errors.

DosSubFreeMem - Remarks

DosSubFreeMem frees a block of memory that was previously allocated by DosSubAllocMem.

cb should be a multiple of 8 bytes, otherwise it will be rounded up. The maximum value for the *cb* parameter is the size of the memory pool initialized by DosSubSetMem minus 64 bytes.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosSubFreeMem - Related Functions

Related Functions

- DosSubAllocMem
- DosSubSetMem
- DosSubUnsetMem

DosSubFreeMem - Example Code

This example creates a heap to manage small amount of memory, and then frees it.

```
#define INCL_DOSMEMMGR
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

#define NUMBLOCKS  128               /* Number of blocks in pool          */
#define POOLSIZE   (NUMBLOCKS*80+64) /* 128 blocks of 80 bytes each.  OS/2
                                        needs 64 bytes to manage the pool */
int main (VOID)
{
  PVOID   pvMemBase       = NULL;       /* Pointer to memory object (pool)*/
  PVOID   apvBlock[NUMBLOCKS] = {NULL}; /* Pointers to memory blocks      */
  ULONG   i               = 0;          /* A loop index                   */
  APIRET  rc              = NO_ERROR;   /* Return code                    */

    /* Allocate the memory pool.  Note the pool is intentionally NOT
       committed here.  OS/2 will commit the memory as needed.          */

  rc = DosAllocMem( &pvMemBase, POOLSIZE, PAG_WRITE );
  if (rc != NO_ERROR) {
    printf("DosAllocMem error:  return code = %u\n", rc);
    return 1;
  }
       /* Make the entire memory object available for Suballocation */

  rc = DosSubSetMem( pvMemBase, DOSSUB_INIT | DOSSUB_SPARSE_OBJ, POOLSIZE );
  if (rc != NO_ERROR) {
    printf("DosSubSetMem error:  return code = %u\n", rc);
    return 1;
  } else { printf("Memory object ready for suballocation.\n"); }

   /* Suballocate the pool into 128 memory blocks, each 80 bytes in size */

  for( i = 0; i < NUMBLOCKS; i++ ) {
    rc = DosSubAllocMem( pvMemBase, &apvBlock[ i ], 80 );
    if( rc != NO_ERROR ) {
      printf( "DosSubAllocMem error: return code = %u  (i = %u)\n", rc, i );
      break;             }
  }

                    /*    USE THE POOL HERE... */
             /* When done, free the 128 memory blocks */

  do {
    rc = DosSubFreeMem(  pvMemBase, apvBlock[ --i ], 80 );
    if (rc != NO_ERROR) {
      printf("DosSubFreeMem error:  return code = %u  (i = %u)\n", rc, i);
      return 1;
    }
  } while( i );

        /* Stop memory block management, and free the memory object */

  rc = DosSubUnsetMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosSubUnsetMem error:  return code = %u\n", rc);
    return 1;
  }

  rc = DosFreeMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosFreeMem error:  return code = %u\n", rc);
    return 1;
  } else {
    printf("Memory freed.\n");
  }
```

```
  return NO_ERROR;
}
```

DosSubFreeMem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSubSetMem

DosSubSetMem - Syntax

Initializes a memory pool for suballocation, or increases the size of a previously initialized memory pool.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pbBase;  /*  The address of the memory pool to be used for suballocation. */
ULONG     flag;    /*  Indicators describing the characteristics of the memory object that is being suballocated
ULONG     cb;      /*  The size, in bytes, of the memory pool. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosSubSetMem(pbBase, flag, cb);
```

DosSubSetMem Parameter - pbBase

pbBase (PVOID) - input
The address of the memory pool to be used for suballocation.

DosSubSetMem Parameter - flag

flag (ULONG) - input
Indicators describing the characteristics of the memory object that is being suballocated.

This parameter consists of the following bit fields:

Bit	Description
0	DOSSUB_INIT (0x00000001) This bit must be set to initialize a memory object for suballocation. Otherwise, the request is to attach a process to a shared memory pool that was previously initialized by another process using DosSubSetMem.
1	DOSSUB_GROW (0x00000002) If this bit is set, then the request is to increase the size of the memory pool being managed. Bit 0 then has no meaning.
2	DOSSUB_SPARSE_OBJ (0x00000004) Bit 2 is set if the requester wants a suballocation function to manage the commitment of the pages spanned by the memory pool. All of the pages spanned by the object must be initially uncommitted. If this bit is clear, the suballocation function assumes that all of the pages spanned by the memory pool are valid and committed. For a DosSubSetMem(Grow) request, the setting of this bit should be the same as when the memory pool was initialized. Otherwise, an error is returned.
3	DOSSUB_SERIALIZE (0x00000008) This bit is set if the requester requires access to the memory pool to be serialized. For shared memory pools, the first DosSubSetMem(Init or Serialize) request causes the memory pool to be created and opened for the requesting process. Subsequent DosSubSetMem(Attach or Serialize) requests cause the shared memory pool to be attached to the requesting process. The requesting process must first gain access to the memory object that the pool resides in. DosSubSetMem(Attach) is indicated when bit 0 is off. On a DosSubSetMem(Grow) request, bit 3 should be the same as when the memory pool was initialized, or an error is returned.

DosSubSetMem Parameter - cb

cb (ULONG) - input
The size, in bytes, of the memory pool.

If the size is not a multiple of 8 bytes, it is rounded down to a multiple of 8.

DosSubSetMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSubSetMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
310	ERROR_DOSSUB_SHRINK

For a full list of error codes, see Errors.

DosSubSetMem - Parameters

pbBase (PVOID) - input
The address of the memory pool to be used for suballocation.

flag (ULONG) - input
Indicators describing the characteristics of the memory object that is being suballocated.

This parameter consists of the following bit fields:

Bit	Description
0	DOSSUB_INIT (0x00000001) This bit must be set to initialize a memory object for suballocation. Otherwise, the request is to attach a process to a shared memory pool that was previously initialized by another process using DosSubSetMem.
1	DOSSUB_GROW (0x00000002) If this bit is set, then the request is to increase the size of the memory pool being managed. Bit 0 then has no meaning.
2	DOSSUB_SPARSE_OBJ (0x00000004) Bit 2 is set if the requester wants a suballocation function to manage the commitment of the pages spanned by the memory pool. All of the pages spanned by the object must be initially uncommitted. If this bit is clear, the suballocation function assumes that all of the pages spanned by the memory pool are valid and committed. For a DosSubSetMem(Grow) request, the setting of this bit should be the same as when the memory pool was initialized. Otherwise, an error is returned.
3	DOSSUB_SERIALIZE (0x00000008) This bit is set if the requester requires access to the memory pool to be serialized. For shared memory pools, the first DosSubSetMem(Init or Serialize) request causes the memory pool to be created and opened for the requesting process. Subsequent DosSubSetMem(Attach or Serialize) requests cause the shared memory pool to be attached to the requesting process. The requesting process must first gain access to the memory object that the pool resides in. DosSubSetMem(Attach) is indicated when bit 0 is off. On a DosSubSetMem(Grow) request, bit 3 should be the same as when the memory pool was initialized, or an error is returned.

cb (ULONG) - input
The size, in bytes, of the memory pool.

If the size is not a multiple of 8 bytes, it is rounded down to a multiple of 8.

ulrc (APIRET) - returns
Return Code.

DosSubSetMem returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
310	ERROR_DOSSUB_SHRINK

For a full list of error codes, see Errors.

DosSubSetMem - Remarks

DosSubSetMem initializes a memory pool for suballocation, or increases the size of a previously initialized memory pool.

The requester must first allocate or gain access to the memory object in which the memory pool resides using one of the memory-management function calls.

All calls to DosSubSetMem must eventually be followed by a call to DosSubUnsetMem. This is necessary to allow the suballocation function

to reset resources used to manage the memory pool.

The size of suballocation control information in the memory pool is 64 bytes. Therefore, the minimum size for DosSubSetMem is 72 bytes.

The requester should not issue DosSetMem or change the attributes of any pages spanned by a memory object that the suballocation function is managing. Otherwise, unpredictable results may occur.

All the pages spanned by the memory pool must have the same attributes. At least Read/Write access must have been requested for the pages spanned by the memory pool when the memory is allocated.

The DosSubSetMem(Grow) function is closely related to the memory and performance requirements of the requester as follows:

- If the requester requires the best performance possible on DosSubAllocMem and DosSubFreeMem functions, and a guarantee that those requests will not fail due to a lack of space on the swap device, the requester should not use the Sparse feature, because the suballocation function will dynamically commit pages and request swap file storage.

 This type of requester may wish to notify the suballocation function later that more committed memory is now available for the memory pool by using the DosSubSetMem(Grow) function.

- Most requesters do not have this kind of requirement. They should allow the suballocation function to manage the pages occupied by the memory pool, and they should initialize it with the Sparse attribute. This type of requester should not have to issue a DosSubSetMem(Grow) function later.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.

- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosSubSetMem - Related Functions

Related Functions

- DosSubAllocMem
- DosSubFreeMem
- DosSubUnsetMem

DosSubSetMem - Example Code

This example creates a heap to manage small amounts of memory, and then frees it.

```
#define INCL_DOSMEMMGR
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

#define NUMBLOCKS  128               /* Number of blocks in pool          */
#define POOLSIZE   (NUMBLOCKS*80+64) /* 128 blocks of 80 bytes each.  OS/2
                                        needs 64 bytes to manage the pool */
int main (VOID)
{
  PVOID   pvMemBase       = NULL;       /* Pointer to memory object (pool)*/
  PVOID   apvBlock[NUMBLOCKS] = {NULL}; /* Pointers to memory blocks      */
  ULONG   i               = 0;          /* A loop index                   */
```

```
  APIRET  rc              = NO_ERROR;   /* Return code                    */

    /* Allocate the memory pool.  Note the pool is intentionally NOT
       committed here.  OS/2 will commit the memory as needed.          */

  rc = DosAllocMem( &pvMemBase, POOLSIZE, PAG_WRITE );
  if (rc != NO_ERROR) {
    printf("DosAllocMem error:  return code = %u\n", rc);
    return 1;
  }
       /* Make the entire memory object available for Suballocation */

  rc = DosSubSetMem( pvMemBase, DOSSUB_INIT | DOSSUB_SPARSE_OBJ, POOLSIZE );
  if (rc != NO_ERROR) {
    printf("DosSubSetMem error:  return code = %u\n", rc);
    return 1;
  } else { printf("Memory object ready for suballocation.\n"); }
   /* Suballocate the pool into 128 memory blocks, each 80 bytes in size */

  for( i = 0; i < NUMBLOCKS; i++ ) {
    rc = DosSubAllocMem( pvMemBase, &apvBlock[ i ], 80 );
    if( rc != NO_ERROR ) {
      printf( "DosSubAllocMem error: return code = %u  (i = %u)\n", rc, i );
      break;             }
  }

                     /*    USE THE POOL HERE... */

              /* When done, free the 128 memory blocks */

  do {
    rc = DosSubFreeMem(  pvMemBase, apvBlock[ --i ], 80 );
    if (rc != NO_ERROR) {
      printf("DosSubFreeMem error:  return code = %u  (i = %u)\n", rc, i);
      return 1;
    }
  } while( i );

        /* Stop memory block management, and free the memory object */

  rc = DosSubUnsetMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosSubUnsetMem error:  return code = %u\n", rc);
    return 1;
  }

  rc = DosFreeMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosFreeMem error:  return code = %u\n", rc);
    return 1;
  } else {
    printf("Memory freed.\n");
  }

  return NO_ERROR;
}
```

DosSubSetMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSubUnsetMem

DosSubUnsetMem - Syntax

Ends the use of a memory pool.

```
#define INCL_DOSMEMMGR
#include <os2.h>

PVOID     pbBase;  /*  The offset of the memory pool whose use is being terminated. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosSubUnsetMem(pbBase);
```

DosSubUnsetMem Parameter - pbBase

pbBase (PVOID) - input
The offset of the memory pool whose use is being terminated.

DosSubUnsetMem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSubUnsetMem returns one of the following values:

0	NO_ERROR
532	ERROR_DOSSUB_CORRUPTED

For a full list of error codes, see Errors.

DosSubUnsetMem - Parameters

pbBase (PVOID) - input
The offset of the memory pool whose use is being terminated.

ulrc (APIRET) - returns
Return Code.

DosSubUnsetMem returns one of the following values:

0	NO_ERROR
532	ERROR_DOSSUB_CORRUPTED

For a full list of error codes, see Errors.

DosSubUnsetMem - Remarks

DosSubUnsetMem ends the use of a memory pool.

All calls to DosSubSetMem must eventually be followed by a call to DosSubUnsetMem.

This call allows the suballocation function to release the resources that it used to manage the suballocation of the memory object. The call to DosSubUnsetMem must occur before the memory object is freed.

Note: DosAllocMem and DosAllocSharedMem both allocate a block of memory of the size requested rounded to the nearest page. On OS/2 Warp, the system allocates a 64K object without attributes on every allocation.

For example, for a DosAllocMem call with a size of 1, the following occurs:

- On OS/2 Warp Connect, the system allocates a 4096-byte block of committed memory.
- On OS/2 Warp, the system allocates a 4096-byte block of committed memory plus 61440 bytes without attributes.

Note:

When you allocate a memory object with the PAG_EXECUTE attribute, it is implied that this memory object also has the PAG_READ attribute. However, when querying the attributes of a memory object, you will get the following results:

- On OS/2 Warp Connect, both PAG_EXECUTE and PAG_READ attributes are returned.
- On OS/2 Warp, only the PAG_EXECUTE attribute is returned. However, PAG_READ is still implied.

DosSubUnsetMem - Related Functions

Related Functions

- DosSubAllocMem
- DosSubFreeMem
- DosSubSetMem

DosSubUnsetMem - Example Code

This example creates a heap to manage small amounts of memory, and then frees it.

```
#define INCL_DOSMEMMGR
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

#define NUMBLOCKS  128               /* Number of blocks in pool          */
#define POOLSIZE   (NUMBLOCKS*80+64) /* 128 blocks of 80 bytes each.  OS/2
                                        needs 64 bytes to manage the pool */
int main (VOID)
{
  PVOID   pvMemBase       = NULL;       /* Pointer to memory object (pool)*/
  PVOID   apvBlock[NUMBLOCKS] = {NULL}; /* Pointers to memory blocks      */
  ULONG   i               = 0;          /* A loop index                   */
  APIRET  rc              = NO_ERROR;   /* Return code                    */

    /* Allocate the memory pool.  Note the pool is intentionally NOT
       committed here.  OS/2 will commit the memory as needed.          */

  rc = DosAllocMem( &pvMemBase, POOLSIZE, PAG_WRITE );
  if (rc != NO_ERROR) {
    printf("DosAllocMem error:  return code = %u\n", rc);
    return 1;
```

```
  }
       /* Make the entire memory object available for Suballocation */

  rc = DosSubSetMem( pvMemBase, DOSSUB_INIT | DOSSUB_SPARSE_OBJ, POOLSIZE );
  if (rc != NO_ERROR) {
    printf("DosSubSetMem error:  return code = %u\n", rc);
    return 1;
  } else { printf("Memory object ready for suballocation.\n"); }

   /* Suballocate the pool into 128 memory blocks, each 80 bytes in size */

  for( i = 0; i < NUMBLOCKS; i++ ) {
    rc = DosSubAllocMem( pvMemBase, &apvBlock[ i ], 80 );
    if( rc != NO_ERROR ) {
      printf( "DosSubAllocMem error: return code = %u  (i = %u)\n", rc, i );
      break;             }
  }

                     /*    USE THE POOL HERE... */

              /* When done, free the 128 memory blocks */

  do {
    rc = DosSubFreeMem(  pvMemBase, apvBlock[ --i ], 80 );
    if (rc != NO_ERROR) {
      printf("DosSubFreeMem error:  return code = %u  (i = %u)\n", rc, i);
      return 1;
    }
  } while( i );
        /* Stop memory block management, and free the memory object */

  rc = DosSubUnsetMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosSubUnsetMem error:  return code = %u\n", rc);
    return 1;
  }

  rc = DosFreeMem( pvMemBase );
  if (rc != NO_ERROR) {
    printf("DosFreeMem error:  return code = %u\n", rc);
    return 1;
  } else {
    printf("Memory freed.\n");
  }

  return NO_ERROR;
}
```

DosSubUnsetMem - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSuppressPopUps

DosSuppressPopUps - Syntax

Suppresses application trap popups and logs them to the file POPUPLOG.OS2.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG     flag;  /*  A flag that indicating whether pop-up suppression should be enabled or disabled. */
CHAR      char;  /*  The drive letter for the log file (used only when pop-up suppression is enabled). */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosSuppressPopUps(flag, char);
```

DosSuppressPopUps Parameter - flag

flag (ULONG) - input
A flag that indicating whether pop-up suppression should be enabled or disabled.

The flag can either be:

SPU_DISABLESUPPRESSION	Disable pop-up suppression.
SPU_ENABLESUPPRESSION	Enable pop-up suppression.

DosSuppressPopUps Parameter - char

char (CHAR) - input
The drive letter for the log file (used only when pop-up suppression is enabled).

DosSuppressPopUps Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSuppressPopUps returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosSuppressPopUps - Parameters

flag (ULONG) - input
A flag that indicating whether pop-up suppression should be enabled or disabled.

The flag can either be:

SPU_DISABLESUPPRESSION	Disable pop-up suppression.
SPU_ENABLESUPPRESSION	Enable pop-up suppression.

char (CHAR) - input
The drive letter for the log file (used only when pop-up suppression is enabled).

ulrc (APIRET) - returns
Return Code.

DosSuppressPopUps returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosSuppressPopUps - Remarks

When pop-ups are suppressed through DosSuppressPopUps, the system does not display trap screens on the user's terminal. The user will not be required to respond to the *abort, retry, continue* message. Instead, the system displays message SYS3571 and logs the name of the process to the POPUPLOG.OS2 log file in the root directory of the specified drive. The system will also log the message inserts in the log file.

If the log file does not exist, the system will create it. If the log file exists, the system appends to it. If the system cannot open the file (for example, because the drive does not exist), the system will not log the pop-up information.

When pop-up suppression is enabled, the system overrides any settings established by DosError on a per-process basis.

To specify the drive for the log file at system initialization, add the following to your CONFIG.SYS file:

```
SUPPRESSPOPUPS=a
```

a , the drive letter, must be a valid upper- or lowercase letter or the system will ignore the line.

The drive letter for the log file can also be specified on the DosSuppressPopUps API.

If an invalid flag (a flag other than SPU_ENABLESUPPRESSION or SPU_DISABLESUPPRESSION) or an invalid drive (a drive other than an upper- or lowercase letter) is specified on DosSuppressPopUps, the system returns error code ERROR_INVALID_PARAMETER. Otherwise, the system returns NO_ERROR.

The following is an example of an entry from the pop-up suppression log:

```
08-19-1994  12:56:31  SYS3188  PID 0007
D:\EXMG\XCPT001Z.EXE
c0000024
1a02c2ea
EAX=00000000  EBX=00040000  ECX=00044854  EDX=00000000
ESI=00044b2c  EDI=000428ec
DS=0053  DSACC=d0f3  DSLIM=1bffffff
ES=0053  ESACC=d0f3  ESLIM=1bffffff
FS=150b  FSACC=00f3  FSLIM=00000030
GS=0000  GSACC=****  GSLIM=********
CS:EIP=005b:1a02c2ea  CSACC=d0df  CSLIM=1bffffff
SS:ESP=0053:0004487c  SSACC=d0f3  SSLIM=1bffffff
EBP=000448c0  FLG=00002202

DOSCALL1.DLL 0002:0000c2ea
```

The log contains information as follows:

- Line one contains:
 - The date and time that the pop-up was written to the log file
 - The message number
 - The PID of the process that ended
- Line two contains the name of the process that ended.

- Subsequent lines display the message inserts, each on a separate line.

In this example, the process was ended due to an unhandled fatal exception, so there are four inserts:

1. Exception number (1 line)
2. Linearized address of faulting EIP (1 line)
3. Register dump (9 lines)
4. Name of module, object number, and offset of faulting EIP (1 line)

DosSuppressPopUps - Related Functions

Related Functions

- DosErrClass
- DosError

DosSuppressPopUps - Example Code

An example of enabling pop-up suppression is:

```
DosSuppressPopUps(SPU_ENABLESUPPRESSION, 'A');
```

An example of disabling pop-up suppression is:

```
DosSuppressPopUps(SPU_DISABLESUPPRESSION, 0);
```

DosSuppressPopUps - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosSuspendThread

DosSuspendThread - Syntax

Temporarily suspends processing of another thread within the current process until DosResumeThread is issued.

```
#define INCL_DOSPROCESS
#include <os2.h>

TID       tid;   /*  Thread identifier of the thread to be suspended. */
APIRET    ulrc;  /*  Return Code. */

ulrc = DosSuspendThread(tid);
```

DosSuspendThread Parameter - tid

tid (TID) - input
Thread identifier of the thread to be suspended.

DosSuspendThread Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosSuspendThread returns one of the following values:

0	NO_ERROR
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosSuspendThread - Parameters

tid (TID) - input
Thread identifier of the thread to be suspended.

ulrc (APIRET) - returns
Return Code.

DosSuspendThread returns one of the following values:

0	NO_ERROR
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosSuspendThread - Remarks

A thread's execution is suspended when another thread in its process issues DosSuspendThread, specifying the ID of the target thread. The thread may not be suspended immediately because it may have locked some system resources that have to be freed first. However, the

thread is not allowed to execute further application program instructions until a corresponding DosResumeThread is issued.

DosSuspendThread permits the suspension of only one other thread within the current process. If a thread needs to disable all thread switching within its process so that the calling thread can execute time-critical code, it issues DosEnterCritSec and DosExitCritSec.

DosKillThread will not terminate a thread that is suspended. Instead the suspended thread will be terminated when it resumes execution. For this reason, you should not kill the main thread of an application if there are any secondary threads that are suspended.

Note: This function is very powerful and must be used with extreme caution. It should be used only when the state of the target thread is known.

DosSuspendThread - Related Functions

Related Functions

- DosCreateThread
- DosEnterCritSec
- DosResumeThread

DosSuspendThread - Example Code

This example creates a new thread within a process, sleeps for 1 second, suspends the thread for 5 seconds, and then waits for the thread to terminate.

Compile this example with MULTITHREAD LIBRARIES. If you are using IBM CSet/2, use the /Gm+ switch.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#define INCL_DOSERRORS      /* DOS error values          */
#include <os2.h>
#include <stdio.h>

void _System CntThreadProc(ULONG LoopMax);  /* Count Thread */

int main(VOID) {
   TID         tidCntThread = 0;     /* ID returned for newly created thread */
   PFNTHREAD   pfnCntThread = &CntThreadProc; /* Address of thread program   */
   ULONG       ulThreadParm = 100;        /* Parameter to thread routine     */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateThread(&tidCntThread,  /* Thread ID (returned by function)  */
                        pfnCntThread,   /* Address of thread program         */
                        ulThreadParm,   /* Parameter passed to ThreadProc    */
                        CREATE_READY |  /* Thread is ready when created      */
                        STACK_SPARSE,   /* Do not pre-commit stack pages     */
                        8192L);         /* Stack size, rounded to page bdy   */
   if (rc != NO_ERROR) {
      printf("DosCreateThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (1000);  /* Sleep for a second to allow thread to run a bit */

   rc = DosSuspendThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosSuspendThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (5000);  /* Sleep 5 seconds before resuming the thread     */

   rc = DosResumeThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosResumeThread error: return code = %u\n", rc);
      return 1;
   }
```

```
   rc = DosWaitThread (&tidCntThread, DCWW_WAIT);
   if (rc != NO_ERROR) {
      printf ("DosWaitThread error : return code = %u\n", rc);
   }

   printf ("Thread has completed!\n");

   return NO_ERROR;
}
void _System CntThreadProc(ULONG LoopMax )  /* Count thread */
{
   ULONG i = 0;                 /* Loop index */

   for (i=0;i < LoopMax;i++ ) {
      printf ("%d\n", i);
   }

   return;
}
```

DosSuspendThread - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosTmrQueryFreq

DosTmrQueryFreq - Syntax

Provides the frequency of the IRQ0 high resolution timer (Intel 8254).

```
#define INCL_DOSPROFILE
#include <os2.h>

PULONG    pulTmrFreq;  /*  Timer Frequency. */
APIRET    rc;          /*  Return Code. */

rc = DosTmrQueryFreq(pulTmrFreq);
```

DosTmrQueryFreq Parameter - pulTmrFreq

pulTmrFreq (PULONG) - output

Timer Frequency.

Provides the frequency of the IRQ0 high resolution timer in hertz (timer units/sec).

DosTmrQueryFreq Return Value - rc

rc (APIRET) - returns
Return Code.

DosTmrQueryFrequency returns the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
535	ERROR_TMR_NO_DEVICE
99	ERROR_DEVICE_IN_USE

For a full list of error codes, see Errors.

DosTmrQueryFreq - Parameters

pulTmrFreq (PULONG) - output
Timer Frequency.

Provides the frequency of the IRQ0 high resolution timer in hertz (timer units/sec).

rc (APIRET) - returns
Return Code.

DosTmrQueryFrequency returns the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
535	ERROR_TMR_NO_DEVICE
99	ERROR_DEVICE_IN_USE

For a full list of error codes, see Errors.

DosTmrQueryFreq - Related Functions

Related Functions

- DosDevConfig
- DosDevIOCtl
- DosPhysicalDisk

DosTmrQueryFreq - Topics

Select an item:

Syntax
Parameters
Returns
Related Functions
Glossary

DosTmrQueryTime

DosTmrQueryTime - Syntax

Provides a snapshot of the time from the IRQ0 high resolution timer (Intel 8254).

```
#define INCL_DOSPROFILE
#include <os2.h>

PQWORD    pqwTmrTime;  /*  Time. */
APIRET    rc;          /*  Return Code. */

rc = DosTmrQueryTime(pqwTmrTime);
```

DosTmrQueryTime Parameter - pqwTmrTime

pqwTmrTime (PQWORD) - output
Time.

A snapshot of the time from the IRQ0 high resolution timer.

DosTmrQueryTime Return Value - rc

rc (APIRET) - returns
Return Code.

DosTmrQueryTime returns the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
535	ERROR_TMR_NO_DEVICE
99	ERROR_DEVICE_IN_USE
536	ERROR_TMR_INVALID_TIME

For a full list of error codes, see Errors.

DosTmrQueryTime - Parameters

pqwTmrTime (PQWORD) - output
Time.

A snapshot of the time from the IRQ0 high resolution timer.

rc (APIRET) - returns
Return Code.

DosTmrQueryTime returns the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
535	ERROR_TMR_NO_DEVICE
99	ERROR_DEVICE_IN_USE
536	ERROR_TMR_INVALID_TIME

For a full list of error codes, see Errors.

DosTmrQueryTime - Related Functions

Related Functions

- DosDevConfig
- DosDevIOCtl
- DosPhysicalDisk

DosTmrQueryTime - Topics

Select an item:
Syntax
Parameters
Returns
Related Functions
Glossary

DosTransactNPipe

DosTransactNPipe - Syntax

Writes to a duplex message pipe, then reads from it.

```
#define INCL_DOSNMPIPES
#include <os2.h>

HPIPE     hpipe;    /*  A named-pipe handle. */
PVOID     pOutbuf;  /*  A pointer to the buffer that is to be written to the pipe. */
```

```
ULONG     cbOut;    /*  The number of bytes to be written. */
PVOID     pInbuf;   /*  A pointer to the buffer for returned data. */
ULONG     cbIn;     /*  The maximum size, in bytes, of returned data. */
PULONG    pcbRead;  /*  A pointer to the number of bytes read. */
APIRET    ulrc;     /*  Return Code. */

ulrc = DosTransactNPipe(hpipe, pOutbuf, cbOut,
         pInbuf, cbIn, pcbRead);
```

DosTransactNPipe Parameter - hpipe

hpipe (HPIPE) - input
A named-pipe handle.

This handle is returned by DosCreateNPipe for a server process, or by DosOpen for a client process.

DosTransactNPipe Parameter - pOutbuf

pOutbuf (PVOID) - input
A pointer to the buffer that is to be written to the pipe.

DosTransactNPipe Parameter - cbOut

cbOut (ULONG) - input
The number of bytes to be written.

DosTransactNPipe Parameter - pInbuf

pInbuf (PVOID) - output
A pointer to the buffer for returned data.

DosTransactNPipe Parameter - cbIn

cbIn (ULONG) - input
The maximum size, in bytes, of returned data.

DosTransactNPipe Parameter - pcbRead

pcbRead (PULONG) - output
A pointer to the number of bytes read.

DosTransactNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosTransactNPipe returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
11	ERROR_BAD_FORMAT
109	ERROR_BROKEN_PIPE
157	ERROR_DISCARDED
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED
234	ERROR_MORE_DATA

For a full list of error codes, see Errors.

DosTransactNPipe - Parameters

hpipe (HPIPE) - input
A named-pipe handle.

This handle is returned by DosCreateNPipe for a server process, or by DosOpen for a client process.

pOutbuf (PVOID) - input
A pointer to the buffer that is to be written to the pipe.

cbOut (ULONG) - input
The number of bytes to be written.

pInbuf (PVOID) - output
A pointer to the buffer for returned data.

cbIn (ULONG) - input
The maximum size, in bytes, of returned data.

pcbRead (PULONG) - output
A pointer to the number of bytes read.

ulrc (APIRET) - returns
Return Code.

DosTransactNPipe returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
11	ERROR_BAD_FORMAT
109	ERROR_BROKEN_PIPE
157	ERROR_DISCARDED
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
233	ERROR_PIPE_NOT_CONNECTED
234	ERROR_MORE_DATA

For a full list of error codes, see Errors.

DosTransactNPipe - Remarks

DosTransactNPipe is intended for use only on a duplex message pipe that is in message-read mode. If this function is issued for a pipe that is not a duplex message pipe, ERROR_BAD_FORMAT is returned.

The current setting of the pipe's blocking mode has no effect on this function; that is, even if the pipe is in nonblocking mode, DosTransactNPipe writes the entire *pOutbuf* to the pipe, and does not return until it reads a response from the pipe into *pInbuf*. If *pInbuf* is too small to contain the response message, ERROR_MORE_DATA is returned.

The function does not succeed if there is any unread data in the pipe, or if the pipe is not in message-read mode.

Clients of named pipes created with the NP_ACCESS_OUTBOUND or NP_ACCESS_INBOUND access mode cannot use the DosTransactNPipe function. If the named pipe's client uses the DosTransactNPipe function, the function returns error code ERROR_ACCESS_DENIED.

An attempt to write to a pipe whose other end has been closed returns ERROR_BROKEN_PIPE or, if the other end was closed without reading all pending data, ERROR_DISCARDED.

DosTransactNPipe - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosCloseMuxWaitSem
- DosConnectNPipe
- DosCreateEventSem
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosWaitNPipe
- DosWaitEventSem
- DosWaitMuxWaitSem
- DosWrite

DosTransactNPipe - Example Code

This example handles the client side of a pipe. It opens an existing named pipe, sends a message to the host, and reads the host reply using the DosTransactNP function.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR         /* DOS File Manager values */
#define INCL_DOSNMPIPES         /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES      /* DOS Semaphore values */
#define INCL_DOSERRORS          /* DOS Error values */
#include <os2.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                     = NO_ERROR;   /* Return code */
   CHAR     outmsg[256]            = "";         /* Output message buffer */
   CHAR     inmsg[256]             = "";         /* Input message buffer */
   HFILE    PipeHandle             = NULLHANDLE; /* Pipe handle */
   PIPEINFO PipeBuffer[4]          = {{0}};
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]            = {0};
   ULONG    bytes                  = 0;
   ULONG    Action                 = 0;
   PIPESEMSTATE infobuf[3]         = {{0}};

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to Pipe.\n");

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);   /* Force above printf to output device */
   gets(outmsg);

   rc = DosTransactNPipe(PipeHandle,          /* Handle of duplex pipe */
                         outmsg,              /* Output message buffer */
                         strlen(outmsg),      /* Size of output message */
                         inmsg,               /* Input message buffer */
                         sizeof(inmsg),       /* SIze of input buffer */
                         &bytes);             /* Number of bytes read */
   if (rc != NO_ERROR) {
      printf("DosTransactNPipe error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST: %s\n\n", inmsg);

   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosTransactNPipe - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosUnsetExceptionHandler

DosUnsetExceptionHandler - Syntax

Removes an exception handler from a thread's chain of exception handlers.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PEXCEPTIONREGISTRATIONRECORD    pERegRec;  /*  A pointer to the exception registration record that describes the
APIRET                          ulrc;      /*  Return Code. */

ulrc = DosUnsetExceptionHandler(pERegRec);
```

DosUnsetExceptionHandler Parameter - pERegRec

pERegRec (PEXCEPTIONREGISTRATIONRECORD) - input
A pointer to the exception registration record that describes the exception handler to be unregistered.

DosUnsetExceptionHandler Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosUnsetExceptionHandler returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosUnsetExceptionHandler - Parameters

pERegRec (PEXCEPTIONREGISTRATIONRECORD) - input
A pointer to the exception registration record that describes the exception handler to be unregistered.

ulrc (APIRET) - returns
Return Code.

DosUnsetExceptionHandler returns one of the following values:

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

For a full list of error codes, see Errors.

DosUnsetExceptionHandler - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosUnsetExceptionHandler deregisters (removes) an exception handler from a thread's chain of registered exception handlers.

For a detailed list of the system exceptions, see System Exceptions.

DosUnsetExceptionHandler - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosExitMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnwindException

DosUnsetExceptionHandler - Example Code

This example sets and unsets an exception handler named "MyTermhandler" from the thread's chain of exception handlers.

```
#define INCL_DOSPROCESS       /* DOS process values (for DosSleep) */
#define INCL_DOSEXCEPTIONS    /* DOS exception values */
#define INCL_ERRORS           /* DOS error values     */
#include <os2.h>
#include <stdio.h>

ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv );

int main (VOID)
{
  EXCEPTIONREGISTRATIONRECORD RegRec  = {0};     /* Exception Registration Record */
  APIRET      rc      = NO_ERROR;   /* Return code                   */

  /* Add MyTermHandler to this thread's chain of exception handlers */

  RegRec.ExceptionHandler = (ERR)MyTermHandler;
  rc = DosSetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosSetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Terminate this program using Ctrl-C or Ctrl-Break.\n");

  rc = DosSleep(60000L);   /* Give user plenty of time to comply... */

  rc = DosUnsetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosUnsetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }
  return NO_ERROR;
}
/*****************************************************************************/
ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv )
{
  printf("*** MyTermHandler entered: ExceptionNum = %x\n",p1->ExceptionNum);

  switch( p1->ExceptionNum) {
```

```
   case XCPT_SIGNAL: {
      printf(" XCPT_SIGNAL");
      switch( p1->ExceptionInfo[0] ) {
        case XCPT_SIGNAL_INTR:     { printf("_INTR"); break; }
        case XCPT_SIGNAL_KILLPROC: { printf("_KILLPROC"); break; }
        case XCPT_SIGNAL_BREAK:    { printf("_BREAK"); break; }
        default:;
      }
      printf("\n"); break;
   }
   case XCPT_PROCESS_TERMINATE:       { printf(" Process Terminate \n"); break;}
   case XCPT_ASYNC_PROCESS_TERMINATE: { printf(" Async Process Term\n"); break;}
   case XCPT_UNWIND:                  { printf(" XCPT_UNWIND\n");        break;}
   default:;
  }

  return XCPT_CONTINUE_SEARCH;          /* Exception not resolved... */
}
```

DosUnsetExceptionHandler - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosUnwindException

DosUnwindException - Syntax

Calls and removes exception handlers from a thread's chain of exception handlers.

```
#define INCL_DOSEXCEPTIONS
#include <os2.h>

PEXCEPTIONREGISTRATIONRECORD    phandler;   /*  A pointer to the exception registration record that describes th
PVOID                           pTargetIP;  /*  A pointer to where DosUnwindException branches after calling all
PEXCEPTIONREPORTRECORD          pERepRec;   /*  An optional pointer to an exception record. */
APIRET                          ulrc;       /*  Return Code. */

ulrc = DosUnwindException(phandler, pTargetIP,
        pERepRec);
```

DosUnwindException Parameter - phandler

phandler (PEXCEPTIONREGISTRATIONRECORD) - input
A pointer to the exception registration record that describes the exception handler to be unregistered.

This parameter can have one of the following values:

Address	A pointer to the exception registration record where the unwind operation should stop.
0	UNWIND_ALL An exit unwind operation is performed. This removes all exception handlers from the thread, and ends the thread.
-1	END_OF_CHAIN All exception handlers for the thread are unwound.

DosUnwindException Parameter - pTargetIP

pTargetIP (PVOID) - input
A pointer to where DosUnwindException branches after calling all applicable handlers.

DosUnwindException Parameter - pERepRec

pERepRec (PEXCEPTIONREPORTRECORD) - input
An optional pointer to an exception record.

Set this field to zero if it is not used.

DosUnwindException Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosUnwindException returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION

For a full list of error codes, see Errors.

DosUnwindException - Parameters

phandler (PEXCEPTIONREGISTRATIONRECORD) - input
A pointer to the exception registration record that describes the exception handler to be unregistered.

This parameter can have one of the following values:

Address	A pointer to the exception registration record where the unwind operation should stop.

0	UNWIND_ALL An exit unwind operation is performed. This removes all exception handlers from the thread, and ends the thread.
-1	END_OF_CHAIN All exception handlers for the thread are unwound.

pTargetIP (PVOID) - input
A pointer to where DosUnwindException branches after calling all applicable handlers.

pERepRec (PEXCEPTIONREPORTRECORD) - input
An optional pointer to an exception record.

Set this field to zero if it is not used.

ulrc (APIRET) - returns
Return Code.

DosUnwindException returns one of the following values:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION

For a full list of error codes, see Errors.

DosUnwindException - Remarks

Note: Do not make Presentation Manager calls from exception handlers.

DosUnwindException "unwinds" (calls and removes) exception handlers from a thread's chain of registered exception handlers. It can unwind up to but not including a specified exception handler, or it can unwind all the exception handlers.

Each exception handler in the linked list from the Thread Information Block (TIB) is called with the unwind bit in the Exception Report Record structure set, indicating an unwind operation. If the call to the exception handler returns, the Exception Registration Record is removed from the linked list, and the next exception handler is processed.

For a detailed list of the system exceptions, see System Exceptions.

DosUnwindException - Related Functions

Related Functions

- DosAcknowledgeSignalException
- DosEnterMustComplete
- DosExitMustComplete
- DosRaiseException
- DosSendSignalException
- DosSetExceptionHandler
- DosSetSignalExceptionFocus
- DosUnsetExceptionHandler

DosUnwindException - Example Code

This example unwinds all handlers. It does not resolve the exception and causes a SYSTEM ERROR pop-up for the exception to be displayed.

```
#define INCL_DOSPROCESS       /* DOS process values (for DosSleep) */
#define INCL_DOSEXCEPTIONS    /* DOS exception values */
```

```
#define INCL_ERRORS           /* DOS error values     */
#include <os2.h>
#include <stdio.h>

ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv );

int main (VOID)
{
  EXCEPTIONREGISTRATIONRECORD RegRec  = {0};     /* Exception Registration Record */
  APIRET      rc      = NO_ERROR;   /* Return code                   */

  /* Add MyTermHandler to this thread's chain of exception handlers */

  RegRec.ExceptionHandler = (ERR)MyTermHandler;
  rc = DosSetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosSetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }

  printf("Terminate this program using Ctrl-C or Ctrl-Break.\n");
  printf("You will get a System Error popup for the exception.\n");

  rc = DosSleep(60000L);   /* Give user plenty of time to comply... */

  rc = DosUnsetExceptionHandler( &RegRec );
  if (rc != NO_ERROR) {
     printf("DosUnsetExceptionHandler error: return code = %u\n",rc);
     return 1;
  }
  return NO_ERROR;
}
/*************************************************************************/
ULONG _System MyTermHandler( PEXCEPTIONREPORTRECORD       p1,
                             PEXCEPTIONREGISTRATIONRECORD p2,
                             PCONTEXTRECORD               p3,
                             PVOID                        pv )
{
  APIRET  rc = NO_ERROR;

  printf("*** MyTermHandler entered: ExceptionNum = %x\n",p1->ExceptionNum);

  rc = DosUnsetExceptionHandler( p2 );  /* Stop recursive entry to handler */
  rc = DosUnwindException( p2,          /* Exception Registration Record */
                           (PVOID) 0,   /* BAD ADDRESS!!! */
                           p1);         /* Exception Report Record */
  if (rc != NO_ERROR) {
     printf("DosUnwindException error:  return code = %u\n", rc);
  }

  return XCPT_CONTINUE_SEARCH;
}
```

DosUnwindException - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosWaitChild

DosWaitChild - Syntax

Places the current thread into a wait state until an asynchronous child process ends. When the process ends, its process identifier, termination code, and result code are returned to the caller.

```
#define INCL_DOSPROCESS
#include <os2.h>

ULONG           action;  /*  An indicator that specifies which process the current thread is waiting to terminat
ULONG           option;  /*  An indicator that specifies whether to return if no child process ends. */
PRESULTCODES    pres;    /*  Address of the structure that contains the termination code and the result code ind
PPID            ppid;    /*  Address of the process identifier of the ending process. */
PID             pid;     /*  Identifier of the process whose termination is being waited for. */
APIRET          ulrc;    /*  Return Code. */

ulrc = DosWaitChild(action, option, pres,
         ppid, pid);
```

DosWaitChild Parameter - action

action (ULONG) - input
An indicator that specifies which process the current thread is waiting to terminate.

The values of this field are as follows:

0 DCWA_PROCESS
The child process indicated by *pid*.

1 DCWA_PROCESSTREE
The child process indicated by *pid* and all of its child processes.

DosWaitChild Parameter - option

option (ULONG) - input
An indicator that specifies whether to return if no child process ends.

The values of this field are as follows:

0 DCWW_WAIT
Wait if no child process ends or until no child processes are outstanding.

1 DCWW_NOWAIT
Do not wait for child processes to end.

DosWaitChild Parameter - pres

pres (PRESULTCODES) - output
Address of the structure that contains the termination code and the result code indicating the reason for the child's termination.

If no process furnishes a result code, the system provides the value -1.

DosWaitChild Parameter - ppid

ppid (PPID) - output
Address of the process identifier of the ending process.

DosWaitChild Parameter - pid

pid (PID) - input
Identifier of the process whose termination is being waited for.

The values of this field are as follows:

0	Any child process. The current thread waits until any child process that was executed with a return code ends, or until there are no more child processes of any type to wait for.
<>0	The indicated child process and all its descendants.

DosWaitChild Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWaitChild returns one of the following values:

0	NO_ERROR
13	ERROR_INVALID_DATA
87	ERROR_INVALID_PARAMETER
128	ERROR_WAIT_NO_CHILDREN
129	ERROR_CHILD_NOT_COMPLETE
184	ERROR_NO_CHILD_PROCESS
303	ERROR_INVALID_PROCID

For a full list of error codes, see Errors.

DosWaitChild - Parameters

action (ULONG) - input
An indicator that specifies which process the current thread is waiting to terminate.

The values of this field are as follows:

0 DCWA_PROCESS
The child process indicated by *pid*.

1 DCWA_PROCESSTREE
The child process indicated by *pid* and all of its child processes.

option (ULONG) - input
An indicator that specifies whether to return if no child process ends.

The values of this field are as follows:

0 DCWW_WAIT
Wait if no child process ends or until no child processes are outstanding.

1 DCWW_NOWAIT
Do not wait for child processes to end.

pres (PRESULTCODES) - output
Address of the structure that contains the termination code and the result code indicating the reason for the child's termination.

If no process furnishes a result code, the system provides the value -1.

ppid (PPID) - output
Address of the process identifier of the ending process.

pid (PID) - input
Identifier of the process whose termination is being waited for.

The values of this field are as follows:

0 Any child process. The current thread waits until any child process that was executed with a return code ends, or until there are no more child processes of any type to wait for.

<>0 The indicated child process and all its descendants.

ulrc (APIRET) - returns
Return Code.

DosWaitChild returns one of the following values:

0	NO_ERROR
13	ERROR_INVALID_DATA
87	ERROR_INVALID_PARAMETER
128	ERROR_WAIT_NO_CHILDREN
129	ERROR_CHILD_NOT_COMPLETE
184	ERROR_NO_CHILD_PROCESS
303	ERROR_INVALID_PROCID

For a full list of error codes, see Errors.

DosWaitChild - Remarks

DosWaitChild waits for completion of a child process whose execution is asynchronous to that of its parent process. The child process is created by DosExecPgm with a value of 2 specified for *execFlag*. If the child process has multiple threads, the result code returned by DosWaitChild is the one passed to it by the DosExit request that ends the process.

DosWaitChild also can wait for the completion of descendant processes of a child process before it returns. However, it does not report status for descendant processes.

If there are no child processes (either active, or ended but with a return code), then DosWaitChild returns an error. If no child processes have ended, DosWaitChild can wait until one ends before returning to the parent process.

To verify that a given return code is from a specific child process, the process identifier must be checked.

To wait for all child processes and descendants to end, it is necessary to:

1. Issue DosWaitChild with a value of 0 for *pid* (wait until *any* child process has ended).
2. When this DosWaitChild returns, issue a DosWaitChild request with *pid* equal to the value returned for *ppid* on the previous

DosWaitChild request, and a value of 1 for *action* (wait for the indicated process and *all* its child processes).

3. Repeat steps 1 and 2 above until the "No child process exists" return code is received.

DosWaitChild will wait for any child processes, regardless of whether or not they were executed with a result code (by calling DosExecPgm with a value of 2 for *execFlag*). DosWaitChild will not return to the caller until a process *with* a return code ends, or until there are no more child processes (of any type) to wait for.

DosWaitChild - Related Functions

Related Functions

- DosExecPgm
- DosExit
- DosKillProcess
- DosKillThread
- DosWaitThread

DosWaitChild - Example Code

This example starts a program called "CHKDSK.COM" with a C: parameter and waits until it finishes. The termination and return codes are displayed at the end.

```
#define INCL_DOSPROCESS       /* Process and thread values */
#define INCL_DOSERRORS        /* DOS error values          */
#define INCL_DOS
#include <os2.h>
#include <stdio.h>

int main(VOID) {
UCHAR       uchLoadError[CCHMAXPATH] = {0}; /* Error info from DosExecPgm */
RESULTCODES ChildRC           = {0};       /* Results from child process  */
PID         pidChild          = 0;         /* pid for child process       */
APIRET      rc                = NO_ERROR;  /* Return code                 */
UCHAR       uchEnvString[14]  = "ANYTHING=HERE\0";  /* Environment string */
UCHAR       uchArgString[14]  = "CHKDSK\0 C: \0";   /* Argument string    */

      /* The argument string consists of the following:
            - the name of the program (followed by a NULL)
            - any parameters supplied to the program (followed by 2 NULLs)
         Only 1 NULL is explicitly specified at the end of this string.
         ASCIIZ strings end with a NULL already, giving us 2 NULLs.       */

  rc = DosExecPgm(uchLoadError,           /* Object name buffer           */
                  sizeof(uchLoadError),   /* Length of object name buffer */
                  EXEC_ASYNCRESULT,       /* Asynchronous/Trace flags     */
                  uchArgString,           /* Argument string              */
                  uchEnvString,           /* Environment string           */
                  &ChildRC,               /* Returns pid of process on an
                                             asynchronous request         */
                  "CHKDSK.COM");          /* Program file name            */

  if (rc != NO_ERROR) {
     printf("DosExecPgm error: return code = %u\n",rc);
     return 1;
  }

  rc = DosWaitChild(DCWA_PROCESS,     /* Look at only the process specified */
                    DCWW_WAIT,        /* Wait until a child terminates      */
                    &ChildRC,         /* Termination codes returned         */
                    &pidChild,        /* pid of terminating process         */
                    ChildRC.codeTerminate);   /* Process (pid) to look at   */

  if (rc != NO_ERROR) {
     printf("DosWaitChild error: return code = %u\n",rc);
     return 1;
  } else {
     printf("Child complete.  Termination Code: %u  Return Code: %u\n",
```

```
            ChildRC.codeTerminate, ChildRC.codeResult);
  }
  return NO_ERROR;
 }
```

DosWaitChild - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosWaitEventSem

DosWaitEventSem - Syntax

Waits for an event semaphore to be posted.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HEV       hev;        /*  The handle of the event semaphore to wait for. */
ULONG     ulTimeout;  /*  The time-out in milliseconds. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosWaitEventSem(hev, ulTimeout);
```

DosWaitEventSem Parameter - hev

hev (HEV) - input
The handle of the event semaphore to wait for.

DosWaitEventSem Parameter - ulTimeout

ulTimeout (ULONG) - input
The time-out in milliseconds.

This is the maximum amount of time the user wants to allow the thread to be blocked.

This parameter can also have the following values:

0	SEM_IMMEDIATE_RETURN DosWaitEventSem returns without blocking the calling thread.
-1	SEM_INDEFINITE_WAIT DosWaitEventSem blocks the calling thread indefinitely.

DosWaitEventSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWaitEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
95	ERROR_INTERRUPT
640	ERROR_TIMEOUT

For a full list of error codes, see Errors.

DosWaitEventSem - Parameters

hev (HEV) - input
The handle of the event semaphore to wait for.

ulTimeout (ULONG) - input
The time-out in milliseconds.

This is the maximum amount of time the user wants to allow the thread to be blocked.

This parameter can also have the following values:

0	SEM_IMMEDIATE_RETURN DosWaitEventSem returns without blocking the calling thread.
-1	SEM_INDEFINITE_WAIT DosWaitEventSem blocks the calling thread indefinitely.

ulrc (APIRET) - returns
Return Code.

DosWaitEventSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
95	ERROR_INTERRUPT
640	ERROR_TIMEOUT

For a full list of error codes, see Errors.

DosWaitEventSem - Remarks

DosWaitEventSem enables a thread to wait for an event semaphore to be posted.

This function can be called by any thread in the process that created the semaphore. Threads in other processes can also call this function, but they must first gain access to the semaphore by calling DosOpenEventSem.

DosWaitEventSem - Related Functions

Related Functions

- DosCloseEventSem
- DosCreateEventSem
- DosOpenEventSem
- DosPostEventSem
- DosQueryEventSem
- DosResetEventSem

DosWaitEventSem - Example Code

This example creates an event semaphore and the asyncronous timer uses it to post when its time expires. Finally, it closes the event semaphore.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {

PSZ     szSemName  = "\\SEM32\\TIMER\\THREAD1\\EVENT1"; /* Semaphore name */
HEV     hevEvent1     = 0;                  /* Event semaphore handle    */
HTIMER  htimerEvent1  = 0;                  /* Timer handle              */
APIRET  rc            = NO_ERROR;           /* Return code               */

   rc = DosCreateEventSem(szSemName,      /* Name of semaphore to create  */
                          &hevEvent1,     /* Handle of semaphore returned */
                          DC_SEM_SHARED,  /* Shared semaphore             */
                          FALSE);         /* Semaphore is in RESET state  */
   if (rc != NO_ERROR) {
       printf("DosCreateEventSem error: return code = %u\n", rc);
       return 1;       }

   rc = DosAsyncTimer(7000L,              /* 7 second interval            */
                      (HSEM) hevEvent1,   /* Semaphore to post            */
                      &htimerEvent1);     /* Timer handler (returned)     */
   if (rc != NO_ERROR) {
       printf("DosAsyncTimer error: return code = %u\n", rc);
       return 1;
   } else {
       printf("Timer will expire in about 7 seconds...\n");
   }

          /* ... add your other processing here... */

   rc = DosWaitEventSem(hevEvent1,            /* Wait for AsyncTimer event */
                (ULONG) SEM_INDEFINITE_WAIT); /* As long as it takes       */
   if (rc != NO_ERROR) {
       printf("DosWaitEventSem error: return code = %u\n", rc);
       return 1;
   }

   rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
   if (rc != NO_ERROR) {
       printf("DosCloseEventSem error: return code = %u", rc);
       return 1;
   }
```

```
return NO_ERROR;
}
```

DosWaitEventSem - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosWaitMuxWaitSem

DosWaitMuxWaitSem - Syntax

Waits for a muxwait semaphore to clear.

```
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMUX      hmux;       /*  The handle of the muxwait semaphore to wait for. */
ULONG     ulTimeout;  /*  The time-out in milliseconds. */
PULONG    pulUser;    /*  A pointer to a ULONG in which a user-defined value is returned. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosWaitMuxWaitSem(hmux, ulTimeout,
         pulUser);
```

DosWaitMuxWaitSem Parameter - hmux

hmux (HMUX) - input
The handle of the muxwait semaphore to wait for.

DosWaitMuxWaitSem Parameter - ulTimeout

ulTimeout (ULONG) - input

The time-out in milliseconds.

This is the maximum amount of time the user wants to allow the thread to be blocked.

This parameter can also have the following values:

0L	SEM_IMMEDIATE_RETURN DosWaitMuxWaitSem returns without blocking the calling thread.
-1L	SEM_INDEFINITE_WAIT DosWaitMuxWaitSem blocks the calling thread indefinitely.

DosWaitMuxWaitSem Parameter - pulUser

pulUser (PULONG) - output
A pointer to a ULONG in which a user-defined value is returned.

The user-defined value is received from the *ulUser* field of the SEMRECORD structure of the semaphore that was posted or released.

If DCMW_WAIT_ANY was specified in the *flAttr* . parameter when the muxwait semaphore was created, this will be the user field of the semaphore that was posted or released. If the muxwait semaphore consists of mutex semaphores, any mutex semaphore that is released is owned by the caller.

If DCMW_WAIT_ALL was specified in the *flAttr* . parameter when the muxwait semaphore was created, this will be the user field of the last semaphore that was posted or released. (If the thread did not block, the last semaphore that was posted or released will also be the last semaphore in the muxwait-semaphore list.) If the muxwait semaphore consists of mutex semaphores, all of the mutex semaphores that are released are owned by the caller.

DosWaitMuxWaitSem Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWaitMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
103	ERROR_TOO_MANY_SEM_REQUESTS
105	ERROR_SEM_OWNER_DIED
286	ERROR_EMPTY_MUXWAIT
287	ERROR_MUTEX_OWNED
292	ERROR_WRONG_TYPE
640	ERROR_TIMEOUT

For a full list of error codes, see Errors.

DosWaitMuxWaitSem - Parameters

hmux (HMUX) - input
The handle of the muxwait semaphore to wait for.

ulTimeout (ULONG) - input

The time-out in milliseconds.

This is the maximum amount of time the user wants to allow the thread to be blocked.

This parameter can also have the following values:

0L	SEM_IMMEDIATE_RETURN DosWaitMuxWaitSem returns without blocking the calling thread.
-1L	SEM_INDEFINITE_WAIT DosWaitMuxWaitSem blocks the calling thread indefinitely.

pulUser (PULONG) - output
A pointer to a ULONG in which a user-defined value is returned.

The user-defined value is received from the *ulUser* field of the SEMRECORD structure of the semaphore that was posted or released.

If DCMW_WAIT_ANY was specified in the *flAttr* . parameter when the muxwait semaphore was created, this will be the user field of the semaphore that was posted or released. If the muxwait semaphore consists of mutex semaphores, any mutex semaphore that is released is owned by the caller.

If DCMW_WAIT_ALL was specified in the *flAttr* . parameter when the muxwait semaphore was created, this will be the user field of the last semaphore that was posted or released. (If the thread did not block, the last semaphore that was posted or released will also be the last semaphore in the muxwait-semaphore list.) If the muxwait semaphore consists of mutex semaphores, all of the mutex semaphores that are released are owned by the caller.

ulrc (APIRET) - returns
Return Code.

DosWaitMuxWaitSem returns one of the following values:

0	NO_ERROR
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
103	ERROR_TOO_MANY_SEM_REQUESTS
105	ERROR_SEM_OWNER_DIED
286	ERROR_EMPTY_MUXWAIT
287	ERROR_MUTEX_OWNED
292	ERROR_WRONG_TYPE
640	ERROR_TIMEOUT

For a full list of error codes, see Errors.

DosWaitMuxWaitSem - Remarks

DosWaitMuxWaitSem enables a thread to wait for a muxwait semaphore to clear.

This function can be issued by any thread in the process that created the semaphore. Threads in other processes can also issue this function, but they must first gain access to the semaphore by issuing DosOpenMuxWaitSem.

DosWaitMuxWaitSem - Related Functions

Related Functions

- DosAddMuxWaitSem
- DosCloseMuxWaitSem
- DosCreateMuxWaitSem
- DosDeleteMuxWaitSem
- DosOpenMuxWaitSem
- DosQueryMuxWaitSem

DosWaitMuxWaitSem - Example Code

This example creates a muxwait semaphore, adds several event semaphores to its list, waits, and finally closes the muxwait semaphore.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX      hmuxHandAny = NULLHANDLE;      /* Muxwaithandle */
 HEV       hev[5]      = {0};             /* Event semaphores */
 SEMRECORD apsr[5]     = {{0}};           /* Semaphore records */
 APIRET    rc          = NO_ERROR;        /* Return code */
 ULONG     ulLoop      = 0;               /* Loop count */
 ULONG     ulSem       = 0;

for (ulLoop = 0; ulLoop < 5; ulLoop++) {
    rc = DosCreateEventSem((PSZ) NULL,
                           &hev[ulLoop],
                           0,
                           FALSE);
    if (rc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n", rc);
      return 1;
    }
    apsr[ulLoop].hsemCur = (HSEM) hev[ulLoop],
    apsr[ulLoop].ulUser = 0;
} /* endfor */

rc = DosCreateMuxWaitSem((PSZ) NULL,
                         &hmuxHandAny,
                         5L,               /* Number of semaphores in list */
                         apsr,             /* Semaphore list */
                         DCMW_WAIT_ANY);   /* Wait for any semaphore */
    if (rc != NO_ERROR) {
      printf("DosCreateMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosWaitMuxWaitSem(hmuxHandAny,
                       SEM_IMMEDIATE_RETURN,
                       &ulSem);           /* No semaphores have been posted, so
                                             we should see a timeout below...    */
    if (rc != ERROR_TIMEOUT) {
      printf("DosWaitMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosDeleteMuxWaitSem(hmuxHandAny, apsr[0].hsemCur);
    if (rc != NO_ERROR) {
      printf("DosDeleteMuxWaitSem error:  return code = %u\n", rc);
      return 1;
    }

rc = DosCloseMuxWaitSem(hmuxHandAny);
     if (rc != NO_ERROR) {
       printf("DosCloseMuxWaitSem error: return code = %u\n", rc);
       return 1;
     }

return NO_ERROR;
}
```

DosWaitMuxWaitSem - Topics

Select an item:
Syntax
Parameters

Returns
Remarks
Example Code
Related Functions
Glossary

DosWaitNPipe

DosWaitNPipe - Syntax

Waits for a named-pipe instance to become available.

```
#define INCL_DOSNMPIPES
#include <os2.h>

PSZ       pszName;  /*  A pointer to the ASCIIZ name of the pipe to be opened. */
ULONG     msec;     /*  The maximum time, in milliseconds, to wait for a named-pipe instance to become available
APIRET    ulrc;     /*  Return Code. */

ulrc = DosWaitNPipe(pszName, msec);
```

DosWaitNPipe Parameter - pszName

pszName (PSZ) - input
A pointer to the ASCIIZ name of the pipe to be opened.

DosWaitNPipe Parameter - msec

msec (ULONG) - input
The maximum time, in milliseconds, to wait for a named-pipe instance to become available.

When a value of 0 is specified, DosWaitNPipe uses the value of *msec* that was specified when the pipe was created with DosCreateNPipe. When a value of -1 is specified, DosWaitNPipe waits indefinitely.

DosWaitNPipe Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWaitNPipe returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
95	ERROR_INTERRUPT
121	ERROR_SEM_TIMEOUT
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY

For a full list of error codes, see Errors.

DosWaitNPipe - Parameters

pszName (PSZ) - input
A pointer to the ASCIIZ name of the pipe to be opened.

msec (ULONG) - input
The maximum time, in milliseconds, to wait for a named-pipe instance to become available.

When a value of 0 is specified, DosWaitNPipe uses the value of *msec* that was specified when the pipe was created with DosCreateNPipe. When a value of -1 is specified, DosWaitNPipe waits indefinitely.

ulrc (APIRET) - returns
Return Code.

DosWaitNPipe returns one of the following values:

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
95	ERROR_INTERRUPT
121	ERROR_SEM_TIMEOUT
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY

For a full list of error codes, see Errors.

DosWaitNPipe - Remarks

DosWaitNPipe enables a client process to wait for a named-pipe instance to become available when all instances are busy. It should be used only when ERROR_PIPE_BUSY is returned from a call to DosOpen.

The *msec* parameter of DosWaitNPipe places a limit on the amount of time the calling process waits for a named-pipe instance to become available, as follows:

- If the time limit is reached before a pipe instance becomes available, ERROR_SEM_TIMEOUT is returned.
- If a time limit of 0 is specified, the system uses the default time-out value that was specified when the pipe was created with DosCreateNPipe.
- If a time limit of -1 is specified, DosWaitNPipe waits indefinitely.

If DosWaitNPipe is successful, the client must again call DosOpen to gain access to the pipe instance.

If more than one client process is blocked on a DosWaitNPipe request, the system gives the next available pipe instance to the process whose thread has the highest priority. If all of the waiting threads have the same priority, the thread that has been waiting the longest receives the next pipe instance.

Note: The priority of a thread can be changed by calling DosSetPriority.

ERROR_BAD_PIPE is returned if you specify an invalid name or file handle.

DosWaitNPipe - Related Functions

Related Functions

- DosCallNPipe
- DosClose
- DosConnectNPipe
- DosCreateNPipe
- DosDisConnectNPipe
- DosDupHandle
- DosOpen
- DosPeekNPipe
- DosQueryNPHState
- DosQueryNPipeInfo
- DosQueryNPipeSemState
- DosRead
- DosResetBuffer
- DosSetNPHState
- DosSetNPipeSem
- DosTransactNPipe
- DosWrite

DosWaitNPipe - Example Code

This example handles the client side of a pipe. It attempts to open an existing named pipe, if the pipe is not available, it uses this API to wait until it becames avalilable and opens it. It then sets the pipe handle state, sends a message to the host, reads the host reply, and finally closes the pipe.

Before running this example, compile and run the example code shown in the DosConnectNPipe, DosCreateNPipe, DosDisConnectNPipe, or DosSetNPipeSem functions, which handles the host side of the pipe.

```
#define INCL_DOSFILEMGR       /* DOS File Manager values */
#define INCL_DOSNMPIPES       /* DOS Named Pipes values */
#define INCL_DOSSEMAPHORES    /* DOS Semaphore values */
#define INCL_DOSERRORS        /* DOS Error values */
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(VOID) {
   APIRET   rc                    = NO_ERROR;   /* Return code */
   CHAR     message[256]          = "";         /* Message buffer */
   HFILE    PipeHandle            = NULLHANDLE; /* Pipe handle */
   struct   _AVAILDATA  BytesAvail = {0};
   UCHAR    Buffer[200]           = {0};
   ULONG    bytes                 = 0;
   ULONG    Action                = 0;
   ULONG    PipeState             = 0;
   ULONG    HandType              = 0;
   ULONG    FlagWord              = 0;
   ULONG    BytesRead             = 0;

   rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                OPEN_FLAGS_FAIL_ON_ERROR, NULL);
   if (rc == ERROR_PIPE_BUSY)
      if (DosWaitNPipe("\\PIPE\\EXAMPLE", 30000)) {
         rc = DosOpen("\\PIPE\\EXAMPLE", &PipeHandle, &Action, 0, 0, FILE_OPEN,
                      OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                      OPEN_FLAGS_FAIL_ON_ERROR, NULL);
      }

   if (rc != NO_ERROR) {
      printf("DosOpen error: error code = %u\n", rc);
      return 1;
   }  else printf("Connected to HOST\n");

   rc = DosSetNPHState(PipeHandle, NP_WAIT);
   if (rc != NO_ERROR) {
```

```
      printf("DosSetNPHState error: error code = %u\n", rc);
      return 1;
   }

   printf("Enter message to send to PIPEHOST: ");

   fflush(NULL);  /* Cause above printf to show on display */
   gets(message);

   rc = DosWrite(PipeHandle, message, strlen(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosWrite error: error code = %u\n", rc);
      return 1;
   }
   rc = DosRead(PipeHandle, message, sizeof(message), &bytes);
   if (rc != NO_ERROR) {
      printf("DosRead error: error code = %u\n", rc);
      return 1;
   }

   printf("\nMessage received from PIPEHOST was: %s\n\n", message);

   rc = DosClose(PipeHandle);
   printf("...Disconnected\n");
   return NO_ERROR;
}
```

DosWaitNPipe - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosWaitThread

DosWaitThread - Syntax

Places the current thread into a wait state until another thread in the current process has ended. It then returns the thread identifier of the ending thread.

```
#define INCL_DOSPROCESS
#include <os2.h>

PTID      ptid;    /*  Address of the thread identification. */
ULONG     option;  /*  An indicator that specifies whether to return if no thread has ended. */
APIRET    ulrc;    /*  Return Code. */

ulrc = DosWaitThread(ptid, option);
```

DosWaitThread Parameter - ptid

ptid (PTID) - in/out
Address of the thread identification.

Input	The address of the ThreadID of the thread of interest. If *ptid* is 0, the current thread waits until the next thread in the process has ended. If *ptid* is nonzero, the current thread waits until the indicated thread has ended.
Output	The ThreadID of the ended thread is returned in this field.

DosWaitThread Parameter - option

option (ULONG) - input
An indicator that specifies whether to return if no thread has ended.

The values of this field are shown in the following list:

0 DCWW_WAIT
The current thread waits until a thread ends. If a thread has already ended, the call returns immediately with the *ptid* .

1 DCWW_NOWAIT
The current thread does not wait if no threads have ended.

DosWaitThread Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWaitThread returns one of the following values:

0	NO_ERROR
95	ERROR_INTERRUPT
294	ERROR_THREAD_NOT_TERMINATED
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosWaitThread - Parameters

ptid (PTID) - in/out
Address of the thread identification.

Input	The address of the ThreadID of the thread of interest. If *ptid* is 0, the current thread waits until the next thread in the process has ended. If *ptid* is nonzero, the current thread waits until the indicated thread has ended.

Output The ThreadID of the ended thread is returned in this field.

option (ULONG) - input
An indicator that specifies whether to return if no thread has ended.

The values of this field are shown in the following list:

0 DCWW_WAIT
The current thread waits until a thread ends. If a thread has already ended, the call returns immediately with the *ptid* .

1 DCWW_NOWAIT
The current thread does not wait if no threads have ended.

ulrc (APIRET) - returns
Return Code.

DosWaitThread returns one of the following values:

0	NO_ERROR
95	ERROR_INTERRUPT
294	ERROR_THREAD_NOT_TERMINATED
309	ERROR_INVALID_THREADID

For a full list of error codes, see Errors.

DosWaitThread - Remarks

DosWaitThread is used to wait for termination of threads within a process. It is usually used so that thread resources (for example, the stack) can be recovered when a thread ends. DosWaitThread waits on any thread within the current process, or on a specific thread within the process, based on the *ptid* parameter's contents. *option* allows the caller the option of waiting until a thread ends, or getting immediate return and status. If no thread has ended and the DCWW_NOWAIT option is specified, the *ptid* field is preserved.

If DosWaitThread is called with the input *ptid* set to the current thread (the thread attempts to wait on its own termination), the ERROR_INVALID_THREADID error code is returned. ERROR_INVALID_THREADID is also returned if a caller attempts to wait on the termination of the thread with a *ptid* of 1.

DosWaitThread - Related Functions

Related Functions

- DosCreateThread
- DosKillThread
- DosWaitChild

DosWaitThread - Example Code

This example creates a new thread within a process, sleeps for 1 second, suspends the thread for 5 seconds, and then waits for the thread to terminate.

Compile this example with MULTITHREAD LIBRARIES. If you are using CSet/2, use the /Gm+ switch.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#define INCL_DOSERRORS      /* DOS error values          */
#include <os2.h>
#include <stdio.h>

void _System CntThreadProc(ULONG LoopMax);  /* Count Thread */
```

```
int main(VOID) {
   TID          tidCntThread = 0;     /* ID returned for newly created thread */
   PFNTHREAD    pfnCntThread = &CntThreadProc; /* Address of thread program   */
   ULONG        ulThreadParm = 100;        /* Parameter to thread routine     */
   APIRET       rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateThread(&tidCntThread,  /* Thread ID (returned by function)  */
                        pfnCntThread,   /* Address of thread program         */
                        ulThreadParm,   /* Parameter passed to ThreadProc    */
                        CREATE_READY |  /* Thread is ready when created      */
                        STACK_SPARSE,   /* Do not pre-commit stack pages     */
                        8192L);         /* Stack size, rounded to page bdy   */
   if (rc != NO_ERROR) {
      printf("DosCreateThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (1000);  /* Sleep for a second to allow thread to run a bit */
   rc = DosSuspendThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosSuspendThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosSleep (5000);  /* Sleep 5 seconds before resuming the thread     */

   rc = DosResumeThread (tidCntThread);
   if (rc != NO_ERROR) {
      printf("DosResumeThread error: return code = %u\n", rc);
      return 1;
   }

   rc = DosWaitThread (&tidCntThread, DCWW_WAIT);
   if (rc != NO_ERROR) {
      printf ("DosWaitThread error : return code = %u\n", rc);
   }

   printf ("Thread has completed!\n");

   return NO_ERROR;
}

void _System CntThreadProc(ULONG LoopMax )  /* Count thread */
{
   ULONG i = 0;                 /* Loop index */

   for (i=0;i < LoopMax;i++ ) {
      printf ("%d\n", i);
   }

   return;
}
```

DosWaitThread - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosWrite

DosWrite - Syntax

Writes a specified number of bytes from a buffer to the specified file.

```
#define INCL_DOSFILEMGR
#include <os2.h>

HFILE     hFile;      /*  File handle from DosOpen. */
PVOID     pBuffer;    /*  Address of the buffer that contains the data to write. */
ULONG     cbWrite;    /*  Number of bytes to write. */
PULONG    pcbActual;  /*  Address of the variable to receive the number of bytes actually written. */
APIRET    ulrc;       /*  Return Code. */

ulrc = DosWrite(hFile, pBuffer, cbWrite, pcbActual);
```

DosWrite Parameter - hFile

hFile (HFILE) - input
File handle from DosOpen.

DosWrite Parameter - pBuffer

pBuffer (PVOID) - input
Address of the buffer that contains the data to write.

DosWrite Parameter - cbWrite

cbWrite (ULONG) - input
Number of bytes to write.

DosWrite Parameter - pcbActual

pcbActual (PULONG) - output
Address of the variable to receive the number of bytes actually written.

DosWrite Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWrite returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK
29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE
112	ERROR_DISK_FULL
157	ERROR_DISCARDED

For a full list of error codes, see Errors.

DosWrite - Parameters

hFile (HFILE) - input
File handle from DosOpen.

pBuffer (PVOID) - input
Address of the buffer that contains the data to write.

cbWrite (ULONG) - input
Number of bytes to write.

pcbActual (PULONG) - output
Address of the variable to receive the number of bytes actually written.

ulrc (APIRET) - returns
Return Code.

DosWrite returns one of the following values:

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK
29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE
112	ERROR_DISK_FULL
157	ERROR_DISCARDED

For a full list of error codes, see Errors.

DosWrite - Remarks

Note: When writing message pipes the application is limited to 64K messages. As well, these messages cannot span 64k boundaries due to the current design of the thunk layer in read or write routines. If the message is not written in an aligned manner, the subsequent read will not be able to handle the messages properly. If a 64k or less message is written to a pipe from an aligned buffer, the read will handle this properly.

DosWrite begins to write at the current file-pointer position. The file pointer is automatically moved by read and write operations. It can be moved to a desired position by issuing DosSetFilePtr.

If the specified file has been opened using the write-through flag, DosWrite writes the data to the disk before returning. Upon return to the caller, *pcbActual* contains the number of bytes actually written.

If there is not enough space on the disk or diskette to write all of the bytes specified by *cbWrite* then DosWrite does not write any bytes. Upon return to the caller, *pcbActual* contains zero.

A value of zero for *cbWrite* is not considered an error. No data transfer occurs, and there is no effect on the file or the file pointer.

If the file is read-only, the write operation to the file is not performed.

If you issue DosOpen with the Direct Open flag set to 1 in the *fsOpenMode* parameter, you have direct access to an entire disk or diskette volume, independent of the file system. You must lock the logical volume before accessing it, and you must unlock the logical volume when you are finished accessing it. Issue DosDevIOCtl for Category 8, DSK_LOCKDRIVE to lock the logical volume, and for Category 8, DSK_UNLOCKDRIVE to unlock the logical volume. While the logical volume is locked, no other process can access it.

Named-Pipe Considerations

DosWrite also is used to write bytes or messages to a named pipe.

Each write operation to a message pipe writes a message whose size is the length of the write operation. DosWrite automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Write operations in blocking mode always write all requested bytes before returning.

In nonblocking mode, DosWrite returns either with all bytes written or none written. DosWrite returns with no bytes written when it would have to divide the message into blocks in order to complete the request. This can occur when there is not enough space left in the pipe, or when the pipe is currently being written to by another client. If this occurs, DosWrite returns immediately with a value of zero for *pcbActual*, indicating that no bytes were written.

For a byte pipe, if the number of bytes to be written exceeds the space available in the pipe, DosWrite writes as many bytes as it can, and returns with the number of bytes actually written in *pcbActual*.

An attempt to write to a pipe whose other end has been closed returns ERROR_BROKEN_PIPE or, if the other end was closed without reading all pending data, ERROR_DISCARDED.

Clients of named pipes created with the NP_ACCESS_OUTBOUND access mode cannot use the DosWrite function. If the named pipe's client uses the DosWrite function, the function returns error code ERROR_ACCESS_DENIED.

DosWrite - Related Functions

Related Functions

- DosOpen
- DosRead
- DosSetFilePtr

DosWrite - Example Code

This example writes a string to a file named "DOSTEST", positions the file pointer back to the beginning of the file, and read the string written. It also opens and closes the file.

```
}
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(void) {
   HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  ulBytesRead    = 0;      /* Number of bytes read by DosRead */
   ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
   ULONG  ulLocal        = 0;      /* File pointer position after DosSetFilePtr */
   UCHAR  uchFileName[20]  = "dostest.dat",     /* Name of file */
```

```
        uchFileData[100] = " ";              /* Data to write to file */
   APIRET rc             = NO_ERROR;         /* Return code */

   /* Open the file test.dat.  Use an existing file or create a new */
   /* one if it doesn't exist.                                      */
   rc = DosOpen(uchFileName,                 /* File path name */
                &hfFileHandle,               /* File handle */
                &ulAction,                   /* Action taken */
                100L,                        /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL, /* File attribute */
                OPEN_ACTION_CREATE_IF_NEW |
                OPEN_ACTION_OPEN_IF_EXISTS,  /* Open function type */
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,       /* Open mode of the file */
                0L);                         /* No extended attribute */

   if (rc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", rc);
      return 1;
   } else {
     printf ("DosOpen: Action taken = %ld\n", ulAction);
   } /* endif */
   /* Write a string to the file */
   strcpy (uchFileData, "testing...\n1...\n2...\n3\n");

   rc = DosWrite (hfFileHandle,              /* File handle */
                  (PVOID) uchFileData,       /* String to be written */
                  sizeof (uchFileData),      /* Size of string to be written */
                  &ulWrote);                 /* Bytes actually written */

   if (rc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosWrite: Bytes written = %u\n", ulWrote);
   } /* endif */

   /* Move the file pointer back to the beginning of the file */
   rc = DosSetFilePtr (hfFileHandle,         /* File Handle */
                       0L,                   /* Offset */
                       FILE_BEGIN,           /* Move from BOF */
                       &ulLocal);            /* New location address */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error: return code = %u\n", rc);
      return 1;
   }

   /* Read the first 100 bytes of the file */
   rc = DosRead (hfFileHandle,              /* File Handle */
                 uchFileData,               /* String to be read */
                 100L,                      /* Length of string to be read */
                 &ulBytesRead);             /* Bytes actually read */

   if (rc != NO_ERROR) {
      printf("DosRead error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosRead: Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
   } /* endif */

   rc = DosClose(hfFileHandle);              /* Close the file */

   if (rc != NO_ERROR) {
      printf("DosClose error: return code = %u\n", rc);
      return 1;
   }
   return NO_ERROR;
}
```

DosWrite - Topics

Select an item:
Syntax

Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

DosWriteQueue

DosWriteQueue - Syntax

Adds an element to a queue.

```
#define INCL_DOSQUEUES
#include <os2.h>

HQUEUE    hq;        /*  The handle of the queue to which an element is to be added. */
ULONG     request;   /*  A value to be passed with the queue element. */
ULONG     cbData;    /*  The length, in bytes, of the data that is being sent to the queue. */
PVOID     pbData;    /*  A pointer to the buffer that contains the data to be placed into the queue. */
ULONG     priority;  /*  The priority value of the element that is being added to the queue. */
APIRET    ulrc;      /*  Return Code. */

ulrc = DosWriteQueue(hq, request, cbData,
         pbData, priority);
```

DosWriteQueue Parameter - hq

hq (HQUEUE) - input
The handle of the queue to which an element is to be added.

DosWriteQueue Parameter - request

request (ULONG) - input
A value to be passed with the queue element.

It is used by an application to code an event. The data is understood by the thread that is adding the element to the queue, as well as by the thread that receives the queue element. There is no special meaning to this data, and the operating system does not alter it.

This value is the same as the data in *pRequest* field of DosPeekQueue operation for examining a queue element.

DosWriteQueue Parameter - cbData

cbData (ULONG) - input
The length, in bytes, of the data that is being sent to the queue.

DosWriteQueue Parameter - pbData

pbData (PVOID) - input
A pointer to the buffer that contains the data to be placed into the queue.

DosWriteQueue Parameter - priority

priority (ULONG) - input
The priority value of the element that is being added to the queue.

This parameter is valid only for queues that were created as priority-based queues, as specified in the *priority* parameter of DosCreateQueue. *priority* is a numerical value in the range of 0 to 15, with 15 being the highest priority.

- If the priority value is 15, the element is added to the top of the queue.
- If the priority value is 0, the element is added as the last element in the queue.
- Elements with the same priority value are grouped together in FIFO (first in, first out) order.

If you assign a value greater than 15 to *priority*, the system sets *priority* to 15. No error code is returned for this condition.

DosWriteQueue Return Value - ulrc

ulrc (APIRET) - returns
Return Code.

DosWriteQueue returns one of the following values:

0	NO_ERROR
334	ERROR_QUE_NO_MEMORY
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosWriteQueue - Parameters

hq (HQUEUE) - input
The handle of the queue to which an element is to be added.

request (ULONG) - input
A value to be passed with the queue element.

It is used by an application to code an event. The data is understood by the thread that is adding the element to the queue, as well as by the thread that receives the queue element. There is no special meaning to this data, and the operating system does not alter it.

This value is the same as the data in *pRequest* field of DosPeekQueue operation for examining a queue element.

cbData (ULONG) - input
The length, in bytes, of the data that is being sent to the queue.

pbData (PVOID) - input
A pointer to the buffer that contains the data to be placed into the queue.

priority (ULONG) - input
The priority value of the element that is being added to the queue.

This parameter is valid only for queues that were created as priority-based queues, as specified in the *priority* parameter of DosCreateQueue. *priority* is a numerical value in the range of 0 to 15, with 15 being the highest priority.

- If the priority value is 15, the element is added to the top of the queue.
- If the priority value is 0, the element is added as the last element in the queue.
- Elements with the same priority value are grouped together in FIFO (first in, first out) order.

If you assign a value greater than 15 to *priority*, the system sets *priority* to 15. No error code is returned for this condition.

ulrc (APIRET) - returns
Return Code.

DosWriteQueue returns one of the following values:

0	NO_ERROR
334	ERROR_QUE_NO_MEMORY
337	ERROR_QUE_INVALID_HANDLE

For a full list of error codes, see Errors.

DosWriteQueue - Remarks

DosWriteQueue adds an element to the specified queue. A client process must request access to the queue by calling DosOpenQueue before it can issue this function. The server process and its threads do not need to issue DosOpenQueue because they already have access to the queue.

If the queue was created as a priority-based queue (as specified in the *priority* parameter of DosCreateQueue), then the priority of the element that is being added must be specified.

If the server process has closed the queue before this request is issued, ERROR_QUE_INVALID_HANDLE is returned.

DosWriteQueue - Related Functions

Related Functions

- DosCloseQueue
- DosCreateQueue
- DosOpenQueue
- DosPeekQueue
- DosPurgeQueue
- DosQueryQueue
- DosReadQueue

DosWriteQueue - Example Code

This example writes to a queue named "SPECIAL.QUE" and reads from it. It also creates, opens, and closes the queue used.

```
#define INCL_DOSQUEUES   /* DOS Queue values */
#define INCL_DOSPROCESS  /* DOS thread and process values */
#define INCL_DOSERRORS   /* DOS error values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

int main(VOID) {

   PSZ         szQueueName  = "\\QUEUES\\SPECIAL.QUE";
   HQUEUE      hqSpecialQue = NULLHANDLE; /* Queue handle                    */
   PSZ         DataBuffer   = "";         /* Data buffer for queue data      */
   REQUESTDATA Request      = {0};        /* Reques */
   PID         pidOwner     = 0;
   ULONG       ulDataLen    = 0;          /* Length of data returned         */
   BYTE        ElemPrty     = 0;          /* Priority of element (returned)  */
   APIRET      rc           = NO_ERROR;   /* Return code                     */

   rc = DosCreateQueue(&hqSpecialQue,    /* Queue handle                     */
            QUE_FIFO |                   /* First-In First-Out order         */
            QUE_CONVERT_ADDRESS,         /* Convert 16-bit addresses to 32   */
            szQueueName);                /* Name of the queue to create      */
   if (rc!= NO_ERROR) {
      printf ("DosCreateQueue error: return code = %u\n", rc);
      return 1;
   }

   rc = DosOpenQueue (&pidOwner,         /* PID of queue owner               */
                      &hqSpecialQue,     /* Handle for created queue         */
                      szQueueName);      /* Name of the queue to open        */
   if (rc!= NO_ERROR) {
      printf ("DosOpenQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "To be, or not to be.  That is the question...";
   rc = DosWriteQueue (hqSpecialQue,        /* Queue to write to           */
                       12345L,              /* Request data                */
                       strlen(DataBuffer),  /* Length of data to write     */
                       DataBuffer,          /* Pointer to data             */
                       0L);              /* Priority (not applicable here) */
   if (rc!= NO_ERROR) {
      printf ("DosWriteQueue error: return code = %u\n", rc);
      return 1;
   }

   DataBuffer = "";                  /* Clear the DataBuffer */
   Request.pid = pidOwner;           /* process ID for the DosReadQueue */
   rc = DosReadQueue (hqSpecialQue,          /* Queue to read from          */
                      &Request,              /* Request data from write     */
                      &ulDataLen,            /* Length of data returned     */
                      (PVOID) &DataBuffer,   /* The data                    */
                      0L,                    /* Remove first element        */
                      DCWW_WAIT,             /* Wait for available data     */
                      &ElemPrty,             /* Priority of data (returned) */
                      0L);             /* Semaphore to use when not waiting */

   if (rc!= NO_ERROR) {
      printf ("DosReadQueue error: return code = %u\n", rc);
      return 1;
   } else {
      printf ("DosReadQueue returns: '%s'\n", DataBuffer);
      printf ("      (Request data = %u)\n", Request.ulData);
   }

   rc = DosCloseQueue(hqSpecialQue);         /* Close the queue */
   if (rc!= NO_ERROR) {
      printf ("DosCloseQueue error: return code = %u\n", rc);
      return 1;
   }

   return NO_ERROR;
}
```

DosWriteQueue - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions
Glossary

Keyboard Functions

This chapter describes the keyboard functions.

The keyboard functions described in this section are:

- KbdCharIn
- KbdFlushBuffer
- KbdGetConsole
- KbdGetCp
- KbdGetHWID
- KbdGetLayout
- KbdGetLayoutUni
- KbdGetStatus
- KbdPeek
- KbdSetCp
- KbdSetLayout
- KbdSetLayoutUni
- KbdSetStatus
- KbdStringIn
- KbdXlate

KbdCharIn

KbdCharIn - Syntax

Reads a character data record from the keyboard.

```
#define INCL_KBD
#include <os2.h>

PKBDKEYINFO     CharData;  /*  Pointer to character data. */
ULONG           Wait;      /*  Wait Flag. */
HKBD            hkbd;      /*  Reserved.  Must be 0. */
APIRET          rc;        /*  Return code. */

rc = KbdCharIn(CharData, Wait, hkbd);
```

KbdCharIn Parameter - CharData

CharData (PKBDKEYINFO) - output
Pointer to character data.

A pointer to a KBDKEYINFO structure in which the character data is returned.

KbdCharIn Parameter - Wait

Wait (ULONG) - input
Wait Flag.

0	IO_WAIT Wait for a keystroke if one is not available. The keystroke returned is removed from the queue.
1	IO_NOWAIT Return immediately, with or without a keystroke. If a keystroke is returned, remove it from the queue.
2	IO_PEEK Return immediately, with or without a keystroke. Do not remove the keystroke from the queue.
3	IO_PEEKWAIT Wait for a keystroke if one is not available. Return the keystroke but do not remove it from the queue.

KbdCharIn Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdCharIn Return Value - rc

rc (APIRET) - returns
Return code.

KbdCharIn returns one of the following values:

0	NO_ERROR
375	ERROR_KBD_INVALID_IOWAIT
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdCharIn - Parameters

CharData (PKBDKEYINFO) - output
Pointer to character data.

A pointer to a KBDKEYINFO structure in which the character data is returned.

Wait (ULONG) - input
Wait Flag.

0	IO_WAIT Wait for a keystroke if one is not available. The keystroke returned is removed from the queue.
1	IO_NOWAIT Return immediately, with or without a keystroke. If a keystroke is returned, remove it from the queue.
2	IO_PEEK Return immediately, with or without a keystroke. Do not remove the keystroke from the queue.
3	IO_PEEKWAIT Wait for a keystroke if one is not available. Return the keystroke but do not remove it from the queue.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdCharIn returns one of the following values:

0	NO_ERROR
375	ERROR_KBD_INVALID_IOWAIT
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdCharIn - Remarks

Note: KbdCharIn returns a complete keystroke. This behavior is unlike the OS/2 1.3 version, which returned only a single byte. This affects only double-byte character set (DBCS) characters.

If bit 0 of *fbStatus* is set, the character returned is either 0 or 0xe0. The Unicode character contains the virtual key.

For valid characters, the character in the current code page is returned, and the Unicode character contains the Unicode encoding of the character.

On an enhanced keyboard, the secondary enter key returns the normal character 0DH and a scan code of E0H.

Extended ASCII codes are identified by bit 1 of the status byte being set to on, and the ASCII character code being either 00H or E0H. Both conditions must be satisfied for the character to be an extended keystroke. For extended ASCII codes, the scan-code byte returned is the second code (extended code). Usually, the extended ASCII code is the scan code of the primary key that was pressed.

A thread in the foreground session that repeatedly polls the keyboard with KbdCharIn (with no wait), can prevent all regular priority-class threads from executing. If polling must be used, and a minimal amount of other processing is being performed, the thread should periodically yield to the CPU by issuing a DosSleep call for an interval of at least 5 milliseconds.

KbdCharIn - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdFlushBuffer

KbdFlushBuffer - Syntax

Removes all entries from the keyboard buffer. This discards all user keystrokes typed ahead.

```
#define INCL_KBD
#include <os2.h>

HKBD      hkbd;    /*  Reserved.  Must be 0. */
APIRET    return;  /*  Return code. */

return = KbdFlushBuffer(hkbd);
```

KbdFlushBuffer Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdFlushBuffer Return Value - return

return (APIRET) - returns
Return code.

KbdFlushBuffer returns one of the following values:

0	NO_ERROR
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdFlushBuffer - Parameters

hkbd (HKBD) - input
Reserved. Must be 0.

return (APIRET) - returns
Return code.

KbdFlushBuffer returns one of the following values:

0	NO_ERROR
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdFlushBuffer - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

KbdGetConsole

KbdGetConsole - Syntax

Reads a key, mouse event, or a notification from the console.

```
#define INCL_KBD
#include <os2.h>

PVOID     Data;    /*  Pointer to event data. */
PULONG    Kind;    /*  Kind of event returned. */
ULONG     Flag;    /*  Wait for a keystroke flag. */
HKBD      hkbd;    /*  Reserved.  Must be 0. */
APIRET    return;  /*  Return code. */

return = KbdGetConsole(Data, Kind, Flag, hkbd);
```

KbdGetConsole Parameter - Data

Data (PVOID) - output
Pointer to event data.

A pointer to a location where the event data is returned. The data type returned depends on the kind of event returned. For keyboard events, the data type returned is KBDKEYINFO. For mouse events, the data type returned is MOUQUEINFO.

KbdGetConsole Parameter - Kind

Kind (PULONG) - output
Kind of event returned.

One of the following values is returned:

0	No event available
1	Keyboard event (not a valid character)
2	Character
3	Mouse event
4	Notification

KbdGetConsole Parameter - Flag

Flag (ULONG) - input
Wait for a keystroke flag.

The possible values are:

0	IO_WAIT Wait for an event if one is not available. The event returned is removed from queue.
1	IO_NOWAIT Return immediately, with or without an event. If an event is returned, removed it from the queue.
2	IO_PEEK Return immediately, with or without an event. Do not remove the event from the queue.
3	IO_PEEKWAIT Wait for an event if one is not available. Return the event but do not remove it from the queue.

KbdGetConsole Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdGetConsole Return Value - return

return (APIRET) - returns
Return code.

KbdGetConsole returns one of the following values:

0	NO_ERROR.
375	ERROR_KBD_INVALID_HANDLE
439	ERROR_KBD_INVALID_IOWAIT
464	ERROR_KBD_DETACHED

KbdGetConsole - Parameters

Data (PVOID) - output

Pointer to event data.

A pointer to a location where the event data is returned. The data type returned depends on the kind of event returned. For keyboard events, the data type returned is KBDKEYINFO. For mouse events, the data type returned is MOUQUEINFO.

Kind (PULONG) - output
Kind of event returned.

One of the following values is returned:

0	No event available
1	Keyboard event (not a valid character)
2	Character
3	Mouse event
4	Notification

Flag (ULONG) - input
Wait for a keystroke flag.

The possible values are:

0	IO_WAIT Wait for an event if one is not available. The event returned is removed from queue.
1	IO_NOWAIT Return immediately, with or without an event. If an event is returned, removed it from the queue.
2	IO_PEEK Return immediately, with or without an event. Do not remove the event from the queue.
3	IO_PEEKWAIT Wait for an event if one is not available. Return the event but do not remove it from the queue.

hkbd (HKBD) - input
Reserved. Must be 0.

return (APIRET) - returns
Return code.

KbdGetConsole returns one of the following values:

0	NO_ERROR.
375	ERROR_KBD_INVALID_HANDLE
439	ERROR_KBD_INVALID_IOWAIT
464	ERROR_KBD_DETACHED

KbdGetConsole - Remarks

KbdGetConsole allows the retrieval of either a keyboard event or mouse event. This should be used by programs using both input devices. See KbdCharIn and MouReadEventQue for details of the returned values and conditions.

Only those mouse events enabled by the event mask are returned. By default, the event mask is disabled.

KbdGetConsole - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdGetCp

KbdGetCp - Syntax

Gets the current keyboard code page.

```
#define INCL_KBD
#include <os2.h>

ULONG      ulReserved;  /*  Reserved.  Must be 0. */
PUSHORT    pidCP;       /*  Pointer to Code-page ID. */
HKBD       hkbd;        /*  Reserved.  Must be 0. */
APIRET     return;      /*  Return code. */

return = KbdGetCp(ulReserved, pidCP, hkbd);
```

KbdGetCp Parameter - ulReserved

ulReserved (ULONG) - input
Reserved. Must be 0.

KbdGetCp Parameter - pidCP

pidCP (PUSHORT) - output
Pointer to Code-page ID.

A pointer to a USHORT in which the code-page ID is returned.

KbdGetCp Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdGetCp Return Value - return

return (APIRET) - returns
Return code.

KbdGetCp returns one of the following values:

0	NO_ERROR.
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdGetCp - Parameters

ulReserved (ULONG) - input
Reserved. Must be 0.

pidCP (PUSHORT) - output
Pointer to Code-page ID.

A pointer to a USHORT in which the code-page ID is returned.

hkbd (HKBD) - input
Reserved. Must be 0.

return (APIRET) - returns
Return code.

KbdGetCp returns one of the following values:

0	NO_ERROR.
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdGetCp - Remarks

The code-page ID is the currently active keyboard code page. A value of 0 indicates the default code-page translation table.

KbdGetCp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdGetHWID

KbdGetHWID - Syntax

Returns the type of keyboard in use. The hardware ID specifies the type of keyboard attached and is defined by the manufacturer.

```
#define INCL_KBD
#include <os2.h>

PKBDHWID    pkbdhwid;  /*  Pointer to hardware ID. */
HKBD        hkbd;      /*  Reserved.  Must be 0. */
APIRET      return;    /*  Return code. */

return = KbdGetHWID(pkbdhwid, hkbd);
```

KbdGetHWID Parameter - pkbdhwid

pkbdhwid (PKBDHWID) - output
Pointer to hardware ID.

A pointer to a KBDHWID structure in which the keyboard hardware ID is returned.

KbdGetHWID Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdGetHWID Return Value - return

return (APIRET) - returns
Return code.

KbdGetHWID returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdGetHWID - Parameters

pkbdhwid (PKBDHWID) - output
Pointer to hardware ID.

A pointer to a KBDHWID structure in which the keyboard hardware ID is returned.

hkbd (HKBD) - input
Reserved. Must be 0.

return (APIRET) - returns
Return code.

KbdGetHWID returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdGetHWID - Remarks

The hardware ID indicates the basic layout of the keyboard, such as whether or not a numeric keypad exists. It gives no information about key values.

KbdGetHWID - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdGetLayout

KbdGetLayout - Syntax

Returns the name of the keyboard layout in use.

```
#define INCL_KBD
#include <os2.h>

PSZ       pszName;  /*  Keyboard layout name. */
HKBD      hkbd;     /*  Reserved.  Must be 0. */
APIRET    rc;       /*  Return code. */

rc = KbdGetLayout(pszName, hkbd);
```

KbdGetLayout Parameter - pszName

pszName (PSZ) - output
Keyboard layout name.

A pointer to the location to return the keyboard layout name. This must be at least 9 bytes long.

KbdGetLayout Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdGetLayout Return Value - rc

rc (APIRET) - returns
Return code.

KbdGetHWID returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdGetLayout - Parameters

pszName (PSZ) - output
Keyboard layout name.

A pointer to the location to return the keyboard layout name. This must be at least 9 bytes long.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdGetHWID returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdGetLayout - Topics

Select an item:
Syntax
Parameters

Returns
Glossary

KbdGetLayoutUni

KbdGetLayoutUni - Syntax

Returns the name of the keyboard layout in use.

```
#define INCL_KBD
#include <os2.h>

USHORT     *name;  /*  Keyboard layout name. */
HKBD        hkbd;  /*  Reserved.  Must be 0. */
APIRET      rc;    /*  Return code. */

rc = KbdGetLayoutUni(name, hkbd);
```

KbdGetLayoutUni Parameter - name

name (USHORT *) - output
Keyboard layout name.

A pointer to the location to return the keyboard layout name in unicode. This must be at least 9 UniChars long.

KbdGetLayoutUni Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdGetLayoutUni Return Value - rc

rc (APIRET) - returns
Return code.

KbdGetHWID returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdGetLayoutUni - Parameters

name (USHORT *) - output
Keyboard layout name.

A pointer to the location to return the keyboard layout name in unicode. This must be at least 9 UniChars long.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdGetHWID returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdGetLayoutUni - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

KbdGetStatus

KbdGetStatus - Syntax

Returns information about the keyboard.

```
#define INCL_KBD
#include <os2.h>

PKBDINFO    pkbdinfo;  /*  Pointer to keyboard data. */
HKBD        hkbd;      /*  Reserved.  Must be 0. */
APIRET      rc;        /*  Return code. */

rc = KbdGetStatus(pkbdinfo, hkbd);
```

KbdGetStatus Parameter - pkbdinfo

pkbdinfo (PKBDINFO) - output
Pointer to keyboard data.

A pointer to a KBDKEYINFO structure in which the keyboard status is returned.

KbdGetStatus Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdGetStatus Return Value - rc

rc (APIRET) - returns
Return code.

KbdGetStatus returns one of the following values:

0	NO_ERROR.
373	ERROR_KBD_PARAMETER
376	ERROR_KBD_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdGetStatus - Parameters

pkbdinfo (PKBDINFO) - output
Pointer to keyboard data.

A pointer to a KBDKEYINFO structure in which the keyboard status is returned.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdGetStatus returns one of the following values:

0	NO_ERROR.
373	ERROR_KBD_PARAMETER
376	ERROR_KBD_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdGetStatus - Remarks

Some of the keyboard status information can be changed using KbdSetStatus.

In KBDINFO, the upper byte of *fsInterim* is the NLS shift state. The meaning of the NLS shift varies by language. The following bits are defined to access this data:

NLSS_NLS1	(0x01) - Full-width, National layer
NLSS_NLS2	(0x02) - Katakana, JAMO phonetic
NLSS_NLS3	(0x04) - Hiragana, Hangeul, TsangJye
NLSS_APPL	(0x10) - Application bit
NLSS_NLS4	(0x40) - Romanji, HanjaCsr
NLSS_KANJI	(0x80) - Kanji, Hanji

KbdGetStatus - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdPeek

KbdPeek - Syntax

Returns a keyboard-character data record, if available, but does not remove it from the queue.

```
#define INCL_KBD
#include <os2.h>

PKBDKEYINFO     CharData;  /*  Pointer to character data. */
HKBD            hkbd;      /*  Reserved.  Must be 0. */
APIRET          rc;        /*  Return code. */

rc = KbdPeek(CharData, hkbd);
```

KbdPeek Parameter - CharData

CharData (PKBDKEYINFO) - output
Pointer to character data.

A pointer to a KBDKEYINFO structure in which the character data is returned.

KbdPeek Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdPeek Return Value - rc

rc (APIRET) - returns
Return code.

KbdPeek returns one of the following values:

0	NO_ERROR.
375	ERROR_KDB_INVALID_IOWAIT
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdPeek - Parameters

CharData (PKBDKEYINFO) - output
Pointer to character data.

A pointer to a KBDKEYINFO structure in which the character data is returned.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdPeek returns one of the following values:

0	NO_ERROR.
375	ERROR_KDB_INVALID_IOWAIT
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdPeek - Remarks

Note: KbdPeek returns a complete keystroke. This behavior is unlike the OS/2 1.3 version, which returned only a single byte. This is significant only for DBCS characters.

If bit 0 of *fbStatus* is set, the character returned is either 0 or 0xe0. The Unicode character contains the virtual key.

For valid characters, the character in the current code page is returned, and the Unicode character contains the Unicode encoding of the character.

KbdPeek - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdSetCp

KbdSetCp - Syntax

Sets the current keyboard code page used to translate keystrokes received from the keyboard. This causes a change in the translation of keys. The code page can be any code page (except EBCDIC to DBCS). Setting this code page does not affect the display or process code pages.

```
#define INCL_KBD
#include <os2.h>

ULONG     ulReserved;  /*  Reserved.  Must be 0. */
USHORT    pidCP;       /*  Code page ID. */
HKBD      hkbd;        /*  Reserved.  Must be 0. */
APIRET    rc;          /*  Return Code. */

rc = KbdSetCp(ulReserved, pidCP, hkbd);
```

KbdSetCp Parameter - ulReserved

ulReserved (ULONG) - input
Reserved. Must be 0.

KbdSetCp Parameter - pidCP

pidCP (USHORT) - input
Code page ID.

The code page ID must be the ID of a code page installed on the system or zero.

KbdSetCp Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdSetCp Return Value - rc

rc (APIRET) - returns
Return Code.

KbdSetCp returns one of the following values:

0
NO_ERROR
439
ERROR_KBD_INVALID_HANDLE
445
ERROR_KBD_FOCUS_REQUIRED
447
ERROR_KBD_KEYBOARD_BUSY
448
ERROR_KBD_INVALID_CODEPAGE
504
ERROR_KBD_EXTENDED_SG

KbdSetCp - Parameters

ulReserved (ULONG) - input
Reserved. Must be 0.

pidCP (USHORT) - input
Code page ID.

The code page ID must be the ID of a code page installed on the system or zero.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return Code.

KbdSetCp returns one of the following values:

0
NO_ERROR
439
ERROR_KBD_INVALID_HANDLE

445
ERROR_KBD_FOCUS_REQUIRED

447
ERROR_KBD_KEYBOARD_BUSY

448
ERROR_KBD_INVALID_CODEPAGE

504
ERROR_KBD_EXTENDED_SG

KbdSetCp - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

KbdSetLayout

KbdSetLayout - Syntax

Allows the keyboard layout to be changed. This change affects only the current session.

```
#define INCL_KBD
#include <os2.h>

PSZ       pszName;  /*  Keyboard layout name. */
HKBD      hkbd;     /*  Reserved.  Must be zero. */
APIRET    rc;       /*  Return code. */

rc = KbdSetLayout(pszName, hkbd);
```

KbdSetLayout Parameter - pszName

pszName (PSZ) - input
Keyboard layout name.

The name of the keyboard layout. The length of the name cannot exceed 8 characters.

KbdSetLayout Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be zero.

KbdSetLayout returns one of the following values:

KbdSetLayout Return Value - rc

rc (APIRET) - returns
Return code.

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdSetLayout - Parameters

pszName (PSZ) - input
Keyboard layout name.

The name of the keyboard layout. The length of the name cannot exceed 8 characters.

hkbd (HKBD) - input
Reserved. Must be zero.

KbdSetLayout returns one of the following values:

rc (APIRET) - returns
Return code.

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdSetLayout - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

KbdSetLayoutUni

KbdSetLayoutUni - Syntax

Allows the keyboard layout to be changed. This change affects only the current session.

```
#define INCL_KBD
#include <os2.h>

USHORT     *name;  /*  Keyboard layout name. */
HKBD        hkbd;  /*  Reserved.  Must be 0. */
APIRET      rc;    /*  Return code. */

rc = KbdSetLayoutUni(name, hkbd);
```

KbdSetLayoutUni Parameter - name

name (USHORT *) - input
Keyboard layout name.

Address of the unicode name of the keyboard layout. The name cannot exceed 8 UniChars in length.

KbdSetLayoutUni Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdSetLayoutUni Return Value - rc

rc (APIRET) - returns
Return code.

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdSetLayoutUni - Parameters

name (USHORT *) - input
Keyboard layout name.

Address of the unicode name of the keyboard layout. The name cannot exceed 8 UniChars in length.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

0	NO_ERROR
373	ERROR_KBD_PARAMETER
439	ERROR_KBD_INVALID_HANDLE

KbdSetLayoutUni - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

KbdSetStatus

KbdSetStatus - Syntax

Sets the characteristics of the keyboard.

```
#define INCL_KBD
#include <os2.h>

PKBDINFO    pkbdinfo;  /*  Pointer to keyboard status. */
HKBD        hkbd;      /*  Reserved.  Must be 0. */
APIRET      rc;        /*  Return code. */

rc = KbdSetStatus(pkbdinfo, hkbd);
```

KbdSetStatus Parameter - pkbdinfo

pkbdinfo (PKBDINFO) - output
Pointer to keyboard status.

A pointer to a KBDINFO structure in which the keyboard status is returned.

KbdSetStatus Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdSetStatus Return Value - rc

rc (APIRET) - returns
Return code.

KbdSetStatus returns one of the following values:

0	NO_ERROR.
376	ERROR_KDB_INVALID_LENGTH
377	ERROR_KBD_INVALID_ECHO_MASK
378	ERROR_KBD_INVALID_INPUT_MASK
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdSetStatus - Parameters

pkbdinfo (PKBDINFO) - output
Pointer to keyboard status.

A pointer to a KBDINFO structure in which the keyboard status is returned.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdSetStatus returns one of the following values:

0	NO_ERROR.
376	ERROR_KDB_INVALID_LENGTH
377	ERROR_KBD_INVALID_ECHO_MASK
378	ERROR_KBD_INVALID_INPUT_MASK
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdSetStatus - Remarks

In the KBDINFO structure, the upper byte of *fsInterim* is the NLS shift state, and can be modified by KbdSetStatus.

KbdSetStatus - Topics

Select an item:
Syntax
Parameters
Returns
Remarks

KbdStringIn

KbdStringIn - Syntax

Gets a string of keyboard input.

```
#define INCL_KBD
#include <os2.h>

PCH             pch;    /*  Pointer to the character string buffer. */
PSTRINGINBUF    pchin;  /*  Pointer to buffer-length data. */
ULONG           Flag;   /*  Wait flag. */
HKBD            hkbd;   /*  Reserved.  Must be 0. */
APIRET          rc;     /*  Return code. */

rc = KbdStringIn(pch, pchin, Flag, hkbd);
```

KbdStringIn Parameter - pch

pch (PCH) - output
Pointer to the character string buffer.

KbdStringIn Parameter - pchin

pchin (PSTRINGINBUF) - in/out
Pointer to buffer-length data.

A pointer to the STRINGINBUF structure containing buffer-length data.

KbdStringIn Parameter - Flag

Flag (ULONG) - input
Wait flag.

0	IO_WAIT Wait for a keystroke if one is not available. The keystroke returned is removed from the queue.
1	IO_NOWAIT

Return immediately, with or without a keystroke. If a keystroke is returned, remove it from the queue.

2 IO_PEEK
Return immediately, with or without a keystroke. Do not remove the keystroke from the queue.

3 IO_PEEKWAIT
Wait for a keystroke if one is not available. Return the keystroke but do not remove it from the queue.

KbdStringIn Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdStringIn Return Value - rc

rc (APIRET) - returns
Return code.

KbdStringIn returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
375	ERROR_KBD_INVALID_IOWAIT
376	ERROR_KBD_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
504	ERROR_KBD_EXTENDED_SG

KbdStringIn - Parameters

pch (PCH) - output
Pointer to the character string buffer.

pchin (PSTRINGINBUF) - in/out
Pointer to buffer-length data.

A pointer to the STRINGINBUF structure containing buffer-length data.

Flag (ULONG) - input
Wait flag.

0 IO_WAIT
Wait for a keystroke if one is not available. The keystroke returned is removed from the queue.

1 IO_NOWAIT
Return immediately, with or without a keystroke. If a keystroke is returned, remove it from the queue.

2 IO_PEEK
Return immediately, with or without a keystroke. Do not remove the keystroke from the queue.

3 IO_PEEKWAIT
Wait for a keystroke if one is not available. Return the keystroke but do not remove it from the queue.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns

Return code.

KbdStringIn returns one of the following values:

0	NO_ERROR
373	ERROR_KBD_PARAMETER
375	ERROR_KBD_INVALID_IOWAIT
376	ERROR_KBD_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
504	ERROR_KBD_EXTENDED_SG

KbdStringIn - Remarks

This function is retained only for compatibility. New code should not use it. KbdGetConsole is the preferred function for new code.

The contents and ending condition depend on the keyboard state:

ASCII	Only valid characters are placed in the input buffer. The function ends when a turnaround character is found or when the buffer is full. The turnaround character is not placed in the buffer.
BINARY	Both valid characters and extended keys are returned. The extended keys are returned as two-byte strings with a leading 0x00 or 0xe0 character. The function ends when the buffer is full.
TERM	Valid characters and escape sequences are placed in the input buffer. The function ends when the turnaround character is found, when an escape sequence is complete, or when the buffer is full. The turnaround character is placed in the buffer.

The maximum buffer length is 255 bytes.

The character strings can be optionally echoed on the display if echo mode is set. When echo is on, each character is echoed as it is read from the keyboard. Echo mode and binary mode are mutually exclusive. See KbdGetStatus and KbdSetStatus for more information.

The default input mode is ASCII. In ASCII mode, 2-byte character codes only return in complete form. An extended ASCII code is returned in a 2-byte string. The first byte is 0DH or E0H and the next byte is an extended code.

In input mode (binary or ASCII), the following returns can be set and retrieved with KbdGetStatus and KbdSetStatus:

Turnaround Character
Echo Mode
Interim Character Flag
Shift State

The received input length is also used by the KbdStringIn line edit functions for re-displaying and entering a caller-specified string. On the next KbdStringIn call the received input length indicates the length of the input buffer that can be recalled by the user using the line-editing keys. A value of 0 inhibits the line-editing function for the current KbdStringIn request.

KbdStringIn - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

KbdXlate

KbdXlate - Syntax

Translates scan codes with shift states into ASCII codes.

```
#define INCL_KBD
#include <os2.h>

PKBDKEYINFO     pKey;  /*  Pointer to character data. */
HKBD            hkbd;  /*  Reserved.  Must be 0. */
APIRET          rc;    /*  Return code. */

rc = KbdXlate(pKey, hkbd);
```

KbdXlate Parameter - pKey

pKey (PKBDKEYINFO) - in/out
Pointer to character data.

A pointer to a KBDKEYINFO structure containing character data.

KbdXlate Parameter - hkbd

hkbd (HKBD) - input
Reserved. Must be 0.

KbdXlate Return Value - rc

rc (APIRET) - returns
Return code.

KbdXlate returns one of the following values:

0	NO_ERROR.
376	ERROR_KDB_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdXlate - Parameters

pKey (PKBDKEYINFO) - in/out
Pointer to character data.

A pointer to a KBDKEYINFO structure containing character data.

hkbd (HKBD) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

KbdXlate returns one of the following values:

0	NO_ERROR.
376	ERROR_KDB_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
445	ERROR_KBD_FOCUS_REQUIRED
447	ERROR_KBD_KEYBOARD_BUSY
504	ERROR_KBD_EXTENDED_SG

KbdXlate - Remarks

KbdXlate is designed for conditions where the scan codes are known, but not the character. This must be used with care, and is not designed to be a substitute for the normal OS/2 keyboard translation functions.

The *bNlsShift* member of KBDKEYINFO must be maintained from call to call whenever an interim bit is set.

KbdXlate - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

Mouse Functions

This chapter describes all of the mouse functions.

The mouse functions described in this section are:

- MouDrawPtr
- MouFlushQue
- MouGetDevStatus
- MouGetEventMask
- MouGetNumButtons
- MouGetNumMickeys
- MouGetNumQueEl
- MouGetPtrPos
- MouGetPtrShape
- MouGetScaleFact
- MouGetThreshold
- MouReadEventQue
- MouRemovePtr
- MouSetDevStatus
- MouSetEventMask

- MouSetPtrPos
- MouSetPtrShape
- MouSetScaleFact
- MouSetThreshold

MouDrawPtr

MouDrawPtr - Syntax

Notifies the mouse device driver that an area previously restricted by MouRemovePtr is now available to the mouse device driver when drawing pointer images.

```
#define INCL_MOU
#include <os2.h>

HMOU      DeviceHandle;  /*  Mouse device handle. */
APIRET    rc;            /*  Return code. */

rc = MouDrawPtr(DeviceHandle);
```

MouDrawPtr Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Must be 0.

MouDrawPtr Return Value - rc

rc (APIRET) - returns
Return code.

MouDrawPtr returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouDrawPtr - Parameters

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouDrawPtr returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouDrawPtr - Remarks

MouDrawPtr is required to begin a pointer-image drawing session. At session start, the collision area (the area restricted from pointer-image drawing) is defined as the size of the display.

After a call to MouDrawPtr, MouRemovePtr can be used to define a new collision area. A subsequent call to MouDrawPtr will cancel the collision area.

MouDrawPtr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouFlushQue

MouFlushQue - Syntax

Directs the mouse device driver to empty the mouse-event queue.

```
#define INCL_MOU
#include <os2.h>

HMOU      DeviceHandle;  /*  Mouse device handle. */
APIRET    rc;            /*  Return code. */

rc = MouFlushQue(DeviceHandle);
```

MouFlushQue Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Must be 0.

MouFlushQue Return Value - rc

rc (APIRET) - returns
Return code.

MouFlushQue returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouFlushQue - Parameters

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouFlushQue returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouFlushQue - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouGetDevStatus

MouGetDevStatus - Syntax

Returns status flags for the installed mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PULONG    DeviceStatus;  /*  Current status flags. */
HMOU      DeviceHandle;  /*  Mouse device handle. */
APIRET    rc;            /*  Return code. */

rc = MouGetDevStatus(DeviceStatus, DeviceHandle);
```

MouGetDevStatus Parameter - DeviceStatus

DeviceStatus (PULONG) - input
Current status flags.

Address of the current status-flag settings for the installed mouse device driver.

The return value is a 4-byte set of bit flags.

Bit	Description
31-10	Reserved. Set to 0.
9	If set, mouse data is returned in mickeys, not pels.
8	If set, the drawing operations for the pointer draw routine are disabled.
7-4	Reserved. Set to 0.
3	If set, the pointer draw routine is disabled by an unsupported mode.
2	If set, a flush is in progress.
1	If set, a block-read is in progress.
0	If set, the event queue is busy with I/O.

MouGetDevStatus Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Set to 0.

MouGetDevStatus Return Value - rc

rc (APIRET) - returns

Return code.

MouDevStatus returns one of the following values:

0	NO_ERROR
385	ERROR_MOUSE_NO_DEVICE
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetDevStatus - Parameters

DeviceStatus (PULONG) - input
Current status flags.

Address of the current status-flag settings for the installed mouse device driver.

The return value is a 4-byte set of bit flags.

Bit	Description
31-10	Reserved. Set to 0.
9	If set, mouse data is returned in mickeys, not pels.
8	If set, the drawing operations for the pointer draw routine are disabled.
7-4	Reserved. Set to 0.
3	If set, the pointer draw routine is disabled by an unsupported mode.
2	If set, a flush is in progress.
1	If set, a block-read is in progress.
0	If set, the event queue is busy with I/O.

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Set to 0.

rc (APIRET) - returns
Return code.

MouDevStatus returns one of the following values:

0	NO_ERROR
385	ERROR_MOUSE_NO_DEVICE
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetDevStatus - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouGetEventMask

MouGetEventMask - Syntax

Returns the current value of the mouse-event queue mask.

```
#define INCL_MOU
#include <os2.h>

PULONG    EventMask;     /*  Event mask word. */
HMOU      DeviceHandle;  /*  Mouse device handle. */
APIRET    rc;            /*  Return code. */

rc = MouGetEventMask(EventMask, DeviceHandle);
```

MouGetEventMask Parameter - EventMask

EventMask (PULONG) - input
Event mask word.

Address in application storage where the event mask of the current mouse device driver is returned to the caller by the mouse device driver.

The EventMask is set by MouSetEventMask and has the following definition:

Bit	Description
31-7	Reserved. Set to 0.
6	Report button-3 press/release events, without mouse motion.
5	Report button-3 press/release events, with mouse motion.
4	Report button-2 press/release events, without mouse motion.
3	Report button-2 press/release events, with mouse motion.
2	Report button-1 press/release events, without mouse motion.
1	Report button-1 press/release events, with mouse motion.
0	Report mouse motion events with no button press/release events.

MouGetEventMask Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Set to 0.

MouGetEventMask Return Value - rc

rc (APIRET) - returns
Return code.

MouGetEventMask returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetEventMask - Parameters

EventMask (PULONG) - input
Event mask word.

Address in application storage where the event mask of the current mouse device driver is returned to the caller by the mouse device driver.

The EventMask is set by MouSetEventMask and has the following definition:

Bit	Description
31-7	Reserved. Set to 0.
6	Report button-3 press/release events, without mouse motion.
5	Report button-3 press/release events, with mouse motion.
4	Report button-2 press/release events, without mouse motion.
3	Report button-2 press/release events, with mouse motion.
2	Report button-1 press/release events, without mouse motion.
1	Report button-1 press/release events, with mouse motion.
0	Report mouse motion events with no button press/release events.

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Set to 0.

rc (APIRET) - returns
Return code.

MouGetEventMask returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetEventMask - Topics

Select an item:
Syntax
Parameters
Returns

MouGetNumButtons

MouGetNumButtons - Syntax

Returns the number of buttons supported on the installed mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PULONG    NumberOfButtons;  /*  Number of mouse buttons. */
HMOU      DeviceHandle;     /*  Mouse device handle. */
APIRET    rc;               /*  Return code. */

rc = MouGetNumButtons(NumberOfButtons, DeviceHandle);
```

MouGetNumButtons Parameter - NumberOfButtons

NumberOfButtons (PULONG) - output
Number of mouse buttons.

Address where the number of physical buttons is to be returned.

The return values for the number of buttons supported are:

Value	Definition
1	One mouse button.
2	Two mouse buttons.
3	Three mouse buttons.

MouGetNumButtons Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Set to 0.

MouGetNumButtons Return Value - rc

rc (APIRET) - returns
Return code.

MouGetNumButtons returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetNumButtons - Parameters

NumberOfButtons (PULONG) - output
Number of mouse buttons.

Address where the number of physical buttons is to be returned.

The return values for the number of buttons supported are:

Value	Definition
1	One mouse button.
2	Two mouse buttons.
3	Three mouse buttons.

DeviceHandle (HMOU) - input
Mouse device handle.

Reserved. Set to 0.

rc (APIRET) - returns
Return code.

MouGetNumButtons returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetNumButtons - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouGetNumMickeys

MouGetNumMickeys - Syntax

Returns the number of mickeys per centimeter for the installed mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PULONG    NumberOfMickeys;  /*  Number of mickeys per centimeter. */
HMOU      DeviceHandle;     /*  Mouse device handle */
APIRET    rc;               /*  Return code. */

rc = MouGetNumMickeys(NumberOfMickeys, DeviceHandle);
```

MouGetNumMickeys Parameter - NumberOfMickeys

NumberOfMickeys (PULONG) - input
Number of mickeys per centimeter.

Address of the number of physical mouse motion units. Mouse motion units are reported in mickeys per centimeter. This value is constant, and is based on the mouse device attached.

MouGetNumMickeys Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Mouse device handle

Reserved. Must be 0.

MouGetNumMickeys Return Value - rc

rc (APIRET) - returns
Return code.

MouGetNumMickeys returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetNumMickeys - Parameters

NumberOfMickeys (PULONG) - input
Number of mickeys per centimeter.

Address of the number of physical mouse motion units. Mouse motion units are reported in mickeys per centimeter. This value is constant, and is based on the mouse device attached.

DeviceHandle (HMOU) - input
Mouse device handle

Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouGetNumMickeys returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetNumMickeys - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouGetNumQueEl

MouGetNumQueEl - Syntax

Returns the current status for the mouse device driver event queue.

```
#define INCL_MOU
#include <os2.h>

PMOUQUEINFO     QueDataRecord;  /*  Pointer to the mouse queue data structure. */
HMOU            DeviceHandle;   /*  Reserved.  Must be 0. */
APIRET          rc;             /*  Return code. */

rc = MouGetNumQueEl(QueDataRecord, DeviceHandle);
```

MouGetNumQueEl Parameter - QueDataRecord

QueDataRecord (PMOUQUEINFO) - output
Pointer to the mouse queue data structure.

MouGetNumQueEl Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouGetNumQueEl Return Value - rc

rc (APIRET) - returns
Return code.

MouGetNumQueEl returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetNumQueEl - Parameters

QueDataRecord (PMOUQUEINFO) - output
Pointer to the mouse queue data structure.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouGetNumQueEl returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetNumQueEl - Remarks

The maximum number of queue elements returned in *cmaxEvents* is established during mouse device-driver configuration.

MouGetNumQueEl - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouGetPtrPos

MouGetPtrPos - Syntax

Queries the mouse device driver to determine the current row and column coordinate position of the mouse pointer.

```
#define INCL_MOU
#include <os2.h>

PPTRLOC     PtrPos;         /*  Pointer to the mouse pointer data structure. */
HMOU        DeviceHandle;   /*  Reserved.  Must be 0. */
APIRET      rc;             /*  Return code. */

rc = MouGetPtrPos(PtrPos, DeviceHandle);
```

MouGetPtrPos Parameter - PtrPos

PtrPos (PPTRLOC) - output
Pointer to the mouse pointer data structure.

MouGetPtrPos Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouGetPtrPos Return Value - rc

rc (APIRET) - returns
Return code.

MouGetPtrPos returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetPtrPos - Parameters

PtrPos (PPTRLOC) - output

Pointer to the mouse pointer data structure.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouGetPtrPos returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetPtrPos - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouGetPtrShape

MouGetPtrShape - Syntax

Gets the pointer shape for the session.

```
#define INCL_MOU
#include <os2.h>

PVOID        PtrBuffer;     /*  Pointer shape buffer. */
PPTRSHAPE    PtrDefRec;     /*  Pointer definition structure. */
HMOU         DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET       rc;            /*  Return code. */

rc = MouGetPtrShape(PtrBuffer, PtrDefRec,
       DeviceHandle);
```

MouGetPtrShape Parameter - PtrBuffer

PtrBuffer (PVOID) - output
Pointer shape buffer.

Address of a buffer containing the bit image used as the pointer shape for the session. The buffer consists of AND and XOR pointer masks.

For CGA-compatible text modes (0, 1, 2, and 3) the following describes the AND and XOR pointer mask bit definitions for each

character cell of the masks. Bit values are*colon.

15	Blinking
14-12	Background color
11	Instensity
10-8	Foreground color
7-0	Character

MouGetPtrShape Parameter - PtrDefRec

PtrDefRec (PPTRSHAPE) - in/out
Pointer definition structure.

Contains information about the pointer shape.

MouGetPtrShape Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouGetPtrShape Return Value - rc

rc (APIRET) - returns
Return code.

MouGetPtrShape returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetPtrShape - Parameters

PtrBuffer (PVOID) - output
Pointer shape buffer.

Address of a buffer containing the bit image used as the pointer shape for the session. The buffer consists of AND and XOR pointer masks.

For CGA-compatible text modes (0, 1, 2, and 3) the following describes the AND and XOR pointer mask bit definitions for each character cell of the masks. Bit values are*colon.

15	Blinking
14-12	Background color
11	Instensity
10-8	Foreground color

7-0	Character

PtrDefRec (PPTRSHAPE) - in/out
Pointer definition structure.

Contains information about the pointer shape.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouGetPtrShape returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetPtrShape - Remarks

The application passes a parameter list with the same meaning as defined for MouSetPtrShape to the mouse device driver. The mouse device driver copies the parameters that describe the pointer shape and attributes into the pointer definition control block pointed to by the *PtrDefRec* parameter. The *cb* field must contain the size in bytes of the application buffer where the device driver is to insert the sessions pointer image. All other fields are returned to the application by MouGetPtrShape.

If the buffer size is insufficient, the *cb* field contains the actual size in bytes of the returned pointer image.

The pointer shape may be set by the application with MouSetPtrShape or may be the default image provided by the installed Pointer Device Driver.

MouGetPtrShape - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouGetScaleFact

MouGetScaleFact - Syntax

Returns a pair of one-word scaling factors for the current mouse device.

```
#define INCL_MOU
#include <os2.h>
```

```
PSCALEFACT    ScaleStruct;   /*  Pointer to a control-block structure. */
HMOU          DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET        rc;            /*  Return code. */

rc = MouGetScaleFact(ScaleStruct, DeviceHandle);
```

MouGetScaleFact Parameter - ScaleStruct

ScaleStruct (PSCALEFACT) - output
Pointer to a control-block structure.

Address of the control-block structure that contains the current row and column coordinate scaling factors. The scaling factors must be greater than or equal to 1, and less than or equal to (32K - 1).

MouGetScaleFact Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouGetScaleFact Return Value - rc

rc (APIRET) - returns
Return code.

MouGetScaleFact returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetScaleFact - Parameters

ScaleStruct (PSCALEFACT) - output
Pointer to a control-block structure.

Address of the control-block structure that contains the current row and column coordinate scaling factors. The scaling factors must be greater than or equal to 1, and less than or equal to (32K - 1).

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouGetScaleFact returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetScaleFact - Remarks

The units of the scale factor depend on the mode of the display screen for the session. If the screen is operating in text mode, the scaling units are relative to characters. If the screen is operating in graphics mode, the scaling units are relative to pels.

MouGetScaleFact - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouGetThreshold

MouGetThreshold - Syntax

Returns the current threshold values.

```
#define INCL_MOU
#include <os2.h>

PTHRESHOLD    Threshold;     /*  Pointer threshold structure. */
HMOU          DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET        rc;            /*  Return code. */

rc = MouGetThreshold(Threshold, DeviceHandle);
```

MouGetThreshold Parameter - Threshold

Threshold (PTHRESHOLD) - input
Pointer threshold structure.

Address of the control block structure that contains the threshold structure.

MouGetThreshold Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouGetThreshold Return Value - rc

rc (APIRET) - returns
Return code.

MouGetThreshold returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetThreshold - Parameters

Threshold (PTHRESHOLD) - input
Pointer threshold structure.

Address of the control block structure that contains the threshold structure.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouGetThreshold returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouGetThreshold - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouReadEventQue

MouReadEventQue - Syntax

Reads an event from the mouse device driver FIFO event queue and places it in a structure provided by the application.

```
#define INCL_MOU
#include <os2.h>

PMOUENVENTINFO    EventMask;     /*  Pointer to the mouse-event queue. */
PULONG            Wait;          /*  Wait flag. */
HMOU              DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET            rc;            /*  Return code. */

rc = MouReadEventQue(EventMask, Wait, DeviceHandle);
```

MouReadEventQue Parameter - EventMask

EventMask (PMOUENVENTINFO) - output
Pointer to the mouse-event queue.

Address of the mouse-event queue data.

MouReadEventQue Parameter - Wait

Wait (PULONG) - input
Wait flag.

A pointer to a ULONG containing the action to take when MouReadEventQue is issued and the mouse event queue is empty. If the mouse event queue is not empty, this parameter is not examined by the mouse support.

May be one of the following values:

0	MOU_NOWAIT Do not wait for data on an empty queue (return a NULL record).
1	MOU_WAIT Wait for data on an empty queue.

MouReadEventQue Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouReadEventQue Return Value - rc

rc (APIRET) - returns
Return code.

MouReadEventQue returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
393	ERROR_MOUSE_NO_DATA
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouReadEventQue - Parameters

EventMask (PMOUENVENTINFO) - output
Pointer to the mouse-event queue.

Address of the mouse-event queue data.

Wait (PULONG) - input
Wait flag.

A pointer to a ULONG containing the action to take when MouReadEventQue is issued and the mouse event queue is empty. If the mouse event queue is not empty, this parameter is not examined by the mouse support.

May be one of the following values:

0	MOU_NOWAIT Do not wait for data on an empty queue (return a NULL record).
1	MOU_WAIT Wait for data on an empty queue.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouReadEventQue returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
393	ERROR_MOUSE_NO_DATA
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouReadEventQue - Remarks

The types of queued events are directly affected by the current value of the event mark. MouSetEventMask is used to indicate the types of events desired, and MouGetEventMask is used to query the current value of the mask. Refer to these functions for more information about the masking of events.

MouReadEventQue - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouRemovePtr

MouRemovePtr - Syntax

Notifies the mouse device driver that the area defined by the passed parameters is for the exclusive use of the application. This area is defined as the collision area and is not available to the mouse device driver when drawing pointer images.

```
#define INCL_MOU
#include <os2.h>

PNOPTRRECT    PtrArea;       /*  Pointer shape collision area. */
HMOU          DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET        rc;            /*  Return code. */

rc = MouRemovePtr(PtrArea, DeviceHandle);
```

MouRemovePtr Parameter - PtrArea

PtrArea (PNOPTRRECT) - input
Pointer shape collision area.

The address of the pointer shape collision area structure.

MouRemovePtr Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouRemovePtr Return Value - rc

rc (APIRET) - returns
Return code.

MouRemovePtr returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouRemovePtr - Parameters

PtrArea (PNOPTRRECT) - input
Pointer shape collision area.

The address of the pointer shape collision area structure.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouRemovePtr returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouRemovePtr - Remarks

MouRemovePtr can be issued by any process in the session. However, only one collision area at a time is active. Each call to MouRemovePtr has the effect of resetting the collision area to the location and area specified by the current call.

If the logical pointer position is outside of the collision area specified by the latest MouRemovePtr call, the pointer image is drawn.

The A MouDrawPtr call effectively cancels a MouRemovePtr call, allowing the pointer to be drawn anywhere on the screen until the next MouRemovePtr call is issued.

MouRemovePtr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouSetDevStatus

MouSetDevStatus - Syntax

Sets the mouse device driver status flags for the installed mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PULONG    DeviceStatus;  /*  Status flags. */
HMOU      DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET    rc;            /*  Return code. */

rc = MouSetDevStatus(DeviceStatus, DeviceHandle);
```

MouSetDevStatus Parameter - DeviceStatus

DeviceStatus (PULONG) - input
Status flags.

The passed parameter is a 4-byte set of flags. Only the high byte has meaning.

Bit	Description
31-10	Reserved. Set to 0.
9	If set, mouse data is returned in mickeys, not pels.
8	If set, the drawing operations for pointer draw routine are disabled.
7-0	Reserved. Set to 0.

MouSetDevStatus Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouSetDevStatus Return Value - rc

rc (APIRET) - returns
Return code.

MouSetDevStatus returns one of the following values:

0	NO_ERROR

387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetDevStatus - Parameters

DeviceStatus (PULONG) - input
Status flags.

The passed parameter is a 4-byte set of flags. Only the high byte has meaning.

Bit	Description
31-10	Reserved. Set to 0.
9	If set, mouse data is returned in mickeys, not pels.
8	If set, the drawing operations for pointer draw routine are disabled.
7-0	Reserved. Set to 0.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouSetDevStatus returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetDevStatus - Remarks

MouSetDevStatus is the complement to MouGetDevStatus. However, not all status flags can be set with MouSetDevStatus. Only the flags corresponding to the following functions can be modified:

- Return data in mickeys.
 - Normally, mouse data is returned to the application with the absolute display mode coordinates of the pointer image position on the display screen. When this status flag is set, mouse data is returned in relative mickeys, a unit of mouse movement.
- Don't call pointer draw device.
 - Normally, the pointer draw device driver is called for all drawing operations. When this status flag is set, the mouse device driver does not call the pointer draw device driver. The application must draw any required pointer image on the screen.

At session initialization, the device status is set to 0.

MouSetDevStatus - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouSetEventMask

MouSetEventMask - Syntax

Assigns a new event mask to the current mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PULONG    EventMask;     /*  Mouse device event mask pointer. */
HMOU      DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET    rc;            /*  Return code. */

rc = MouSetEventMask(EventMask, DeviceHandle);
```

MouSetEventMask Parameter - EventMask

EventMask (PULONG) - input
Mouse device event mask pointer.

The EventMask bit values are:

Bit	Description
31-7	Reserved. Set to 0.
6	Report button 3 press/release events, without mouse motion.
5	Report button 3 press/release events, with mouse motion.
4	Report button 2 press/release events, without mouse motion.
3	Report button 2 press/release events, with mouse motion.
2	Report button 1 press/release events, without mouse motion.
1	Report button 1 press/release events, with mouse motion.
0	Mouse motion events with no button press/release events.

MouSetEventMask Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouSetEventMask Return Value - rc

rc (APIRET) - returns
Return code.

MouSetEventMask returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetEventMask - Parameters

EventMask (PULONG) - input
Mouse device event mask pointer.

The EventMask bit values are:

Bit	Description
31-7	Reserved. Set to 0.
6	Report button 3 press/release events, without mouse motion.
5	Report button 3 press/release events, with mouse motion.
4	Report button 2 press/release events, without mouse motion.
3	Report button 2 press/release events, with mouse motion.
2	Report button 1 press/release events, without mouse motion.
1	Report button 1 press/release events, with mouse motion.
0	Mouse motion events with no button press/release events.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouSetEventMask returns one of the following values:

0	NO_ERROR
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetEventMask - Remarks

Setting a bit in the event mask means that the associated event is reported on the mouse FIFO event queue.

At session initialization, the event mask is set to report all events.

MouSetEventMask - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

MouSetPtrPos

MouSetPtrPos - Syntax

Directs the mouse driver to set a new row and column coordinate position for the mouse pointer.

```
#define INCL_MOU
#include <os2.h>

PPTRLOC    PtrPos;        /*  Pointer position. */
HMOU       DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET     rc;            /*  Return code. */

rc = MouSetPtrPos(PtrPos, DeviceHandle);
```

MouSetPtrPos Parameter - PtrPos

PtrPos (PPTRLOC) - input
Pointer position.

Address of the mouse pointer position structure.

MouSetPtrPos Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouSetPtrPos Return Value - rc

rc (APIRET) - returns
Return code.

MouSetPtrPos returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetPtrPos - Parameters

PtrPos (PPTRLOC) - input
Pointer position.

Address of the mouse pointer position structure.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouSetPtrPos returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetPtrPos - Remarks

The application must ensure that the specified coordinate position conforms to the current display mode orientation for the session. Pel values must be used for graphics modes, and character values must be used for text modes.

This function has no effect on the definition of the display's current collision area specified in MouDrawPtr. If the mouse pointer image is directed into a defined collision area, the pointer image is not drawn until either the pointer is moved outside the collision area, or the collision area is released by MouDrawPtr.

At session initialization, the pointer is set to the center of the screen but is not drawn, because the collision area is set to the entire screen.

MouSetPtrPos - Topics

Select an item:
Syntax

Parameters
Returns
Remarks
Glossary

MouSetPtrShape

MouSetPtrShape - Syntax

Sets the pointer shape and size to be used as the mouse device driver pointer image for all applications in a session.

```
#define INCL_MOU
#include <os2.h>

PBYTE        PtrBuffer;     /*  Pointer-shape buffer. */
PPTRSHAPE    PtrDefRec;     /*  Pointer definition record. */
HMOU         DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET       rc;            /*  Return code. */

rc = MouSetPtrShape(PtrBuffer, PtrDefRec,
       DeviceHandle);
```

MouSetPtrShape Parameter - PtrBuffer

PtrBuffer (PBYTE) - input
Pointer-shape buffer.

Address of a buffer containing the bit image used as the pointer shape for the session. The buffer consists of AND and XOR pointer masks.

For CGA-compatible text modes (0, 1, 2, and 3) the following describes the AND and XOR pointer mask bit definitions for each character cell of the masks. Bit values are*colon.

15	Blinking
14-12	Background color
11	Instensity
10-8	Foreground color
7-0	Character

MouSetPtrShape Parameter - PtrDefRec

PtrDefRec (PPTRSHAPE) - input
Pointer definition record.

Address of the structure where the application stores the necessary data for the pointer draw device driver to build a row-by-column image for each bit plane for the current display mode.

MouSetPtrShape Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouSetPtrShape Return Value - rc

rc (APIRET) - returns
Return code.

MouSetPtrShape returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetPtrShape - Parameters

PtrBuffer (PBYTE) - input
Pointer-shape buffer.

Address of a buffer containing the bit image used as the pointer shape for the session. The buffer consists of AND and XOR pointer masks.

For CGA-compatible text modes (0, 1, 2, and 3) the following describes the AND and XOR pointer mask bit definitions for each character cell of the masks. Bit values are*colon.

15	Blinking
14-12	Background color
11	Instensity
10-8	Foreground color
7-0	Character

PtrDefRec (PPTRSHAPE) - input
Pointer definition record.

Address of the structure where the application stores the necessary data for the pointer draw device driver to build a row-by-column image for each bit plane for the current display mode.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

MouSetPtrShape returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED

MouSetPtrShape - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

MouSetScaleFact

MouSetScaleFact - Syntax

Assigns a new set of scaling factors to the current mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PSCALEFACT    ScaleStruct;   /*  Address of scaling factors. */
HMOU          DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET        rc;            /*  Return code */

rc = MouSetScaleFact(ScaleStruct, DeviceHandle);
```

MouSetScaleFact Parameter - ScaleStruct

ScaleStruct (PSCALEFACT) - input
Address of scaling factors.

The address of the control block structure that contains the current row-and-column coordinate scaling factors. The scaling factors must be greater than or equal to 1 and less than or equal to (32K-1).

MouSetScaleFact Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouSetScaleFact Return Value - rc

rc (APIRET) - returns
Return code

MouSetScaleFact returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetScaleFact - Parameters

ScaleStruct (PSCALEFACT) - input
Address of scaling factors.

The address of the control block structure that contains the current row-and-column coordinate scaling factors. The scaling factors must be greater than or equal to 1 and less than or equal to (32K-1).

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code

MouSetScaleFact returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetScaleFact - Remarks

MouSetScaleFact sets the mickey-to-pixel ratio for mouse motion. The row scale and column scale ratios specify a number of mickeys for each 8 pixels. The default value for the row scale is 16 mickeys for each 8 pixels. The default value for the column scale is 8 mickeys to 8 pixels.

The number of pixels moved does not have to be one-to-one for the number of mickeys the mouse moves. The scaling factor defines a sensitivity for the mouse that is a ratio of the number of mickeys required to move the cursor 8 pixels on the screen. The sensitivity determines at what rate the cursor moves on the screen.

MouSetScaleFact - Topics

Select an item:
Syntax
Parameters
Returns

Remarks
Glossary

MouSetThreshold

MouSetThreshold - Syntax

Assigns a new set of threshold values to the current mouse device driver.

```
#define INCL_MOU
#include <os2.h>

PTHRESHOLD    Threshold;     /*  Threshold structure. */
HMOU          DeviceHandle;  /*  Reserved.  Must be 0. */
APIRET        rc;            /*  Return code */

rc = MouSetThreshold(Threshold, DeviceHandle);
```

MouSetThreshold Parameter - Threshold

Threshold (PTHRESHOLD) - input
Threshold structure.

The address of the control block structure that contains the threshold structure.

MouSetThreshold Parameter - DeviceHandle

DeviceHandle (HMOU) - input
Reserved. Must be 0.

MouSetThreshold Return Value - rc

rc (APIRET) - returns
Return code

MouSetThreshold returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED

501	ERROR_MOUSE_NO_CONSOLE

MouSetThreshold - Parameters

Threshold (PTHRESHOLD) - input
Threshold structure.

The address of the control block structure that contains the threshold structure.

DeviceHandle (HMOU) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code

MouSetThreshold returns one of the following values:

0	NO_ERROR
387	ERROR_MOUSE_INV_PARMS
466	ERROR_MOU_DETACHED
501	ERROR_MOUSE_NO_CONSOLE

MouSetThreshold - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

Video Functions

This chapter describes the video functions.

The video functions described in this section are:

- VioAssociate
- VioCreateLogFont
- VioCreatePS
- VioDeleteSetId
- VioDestroyPS
- VioEndPopUp
- VioGetAnsi
- VioGetBuf
- VioGetConfig
- VioGetCp
- VioGetCurPos
- VioGetCurType
- VioGetDeviceCellSize
- VioGetMode
- VioGetOrigin
- VioGetState
- VioModeUndo
- VioModeWait
- VioPopUp
- VioQueryFonts
- VioQuerySetIds

- VioReadCellStr
- VioReadCharStr
- VioSavRedrawWait
- VioSavRedrawUndo
- VioScrollDown
- VioScrLock
- VioScrollLeft
- VioScrollRight
- VioScrollUp
- VioScrUnLock
- VioSetAnsi
- VioSetCp
- VioSetCurPos
- VioSetCurType
- VioSetDeviceCellSize
- VioSetMode
- VioSetOrigin
- VioSetState
- VioShowBuf
- VioShowPS
- VioWrtCellStr
- VioWrtCharStr
- VioWrtCharStrAtt
- VioWrtNAttr
- VioWrtNCell
- VioWrtNChar
- VioWrtTTY

VioAssociate

VioAssociate - Syntax

Associates or disassociates a VIO presentation space with a device context.

```
#define INCL_VIO
#include <os2.h>

HDC       hdc;   /*  Device-context handle. */
HVIO      hvps;  /*  VIO presentation space handle. */
APIRET    rc;    /*  Return code. */

rc = VioAssociate(hdc, hvps);
```

VioAssociate Parameter - hdc

hdc (HDC) - input
Device-context handle.

If this is NULL, a disassociation occurs.

VioAssociate Parameter - hvps

hvps (HVIO) - input
VIO presentation space handle.

This is returned from VioCreatePS.

VioAssociate Return Value - rc

rc (APIRET) - returns
Return code.

VioAssociate returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
495	ERROR_VIO_NOT_PRES_MGR_SG
499	ERROR_VIO_ASSOCIATED_DC

VioAssociate - Parameters

hdc (HDC) - input
Device-context handle.

If this is NULL, a disassociation occurs.

hvps (HVIO) - input
VIO presentation space handle.

This is returned from VioCreatePS.

rc (APIRET) - returns
Return code.

VioAssociate returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
495	ERROR_VIO_NOT_PRES_MGR_SG
499	ERROR_VIO_ASSOCIATED_DC

VioAssociate - Remarks

Subsequent VIO calls to this VIO presentation space will direct output to the specified device context.

If a null handle is specified for the device context, the presentation space is disassociated from any device context.

An associated presentation space or device context cannot be associated.

The screen device context is the only kind of device that can be associated with a VIO presentation space.

VioAssociate - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioCreateLogFont

VioCreateLogFont - Syntax

Specifies a font for use by a VIO session.

```
#define INCL_VIO
#include <os2.h>

PFATTRS    attrs;  /*  Pointer to the font attribute structure. */
ULONG      lcid;   /*  Local identifier. */
STR8       Name;   /*  Logical font name. */
HVIO       hvps;   /*  VIO presentation-space handle. */
APIRET     rc;     /*  Return code. */

rc = VioCreateLogFont(attrs, lcid, Name, hvps);
```

VioCreateLogFont Parameter - attrs

attrs (PFATTRS) - input
Pointer to the font attribute structure.

VioCreateLogFont Parameter - lcid

lcid (ULONG) - input
Local identifier.

This must be a value in the range 1 to 3. It is an error if *lcid* is already in use.

VioCreateLogFont Parameter - Name

Name (STR8) - input
Logical font name.

This string is optional and is used to describe the logical font.

VioCreateLogFont Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

VioCreateLogFont Return Value - rc

rc (APIRET) - returns
Return code.

VioCreateLogFont returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioCreateLogFont - Parameters

attrs (PFATTRS) - input
Pointer to the font attribute structure.

lcid (ULONG) - input
Local identifier.

This must be a value in the range 1 to 3. It is an error if *lcid* is already in use.

Name (STR8) - input
Logical font name.

This string is optional and is used to describe the logical font.

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioCreateLogFont returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioCreateLogFont - Remarks

The system selects the font most closely matching the specified font from the set of monospaced fonts installed in the system.

In OS/2 2.x, *hvps* cannot be 0.

VioCreateLogFont - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioCreatePS

VioCreatePS - Syntax

Creates a VIO presentation space.

```
#define INCL_VIO
#include <os2.h>

PHVIO     phvps;      /*  Pointer to the presentation-space handle. */
ULONG     Rows;       /*  Number of rows. */
ULONG     Columns;    /*  Number of columns. */
ULONG     Format;     /*  Format of the attributes */
ULONG     AttrBytes;  /*  Number of attribute bytes. */
HVIO      hvps;       /*  Reserved.  Must be 0. */
APIRET    rc;         /*  Return code. */

rc = VioCreatePS(phvps, Rows, Columns, Format,
       AttrBytes, hvps);
```

VioCreatePS Parameter - phvps

phvps (PHVIO) - output
Pointer to the presentation-space handle.

The location where the newly created presentation-space handle is to be returned.

VioCreatePS Parameter - Rows

Rows (ULONG) - input
Number of rows.

The number of rows in the presentation space. The maximum value allowed is 255.

VioCreatePS Parameter - Columns

Columns (ULONG) - input
Number of columns.

The number of columns in the presentation space. The maximum value allowed is 255.

VioCreatePS Parameter - Format

Format (ULONG) - input
Format of the attributes

The attributes may be one of the following format types:

1	VGA compatible
2	Unicode
3	MFI compatible

VioCreatePS Parameter - AttrBytes

AttrBytes (ULONG) - input
Number of attribute bytes.

This is used along with the format to select the attribute structure. This field has a value of 1, 2, or 3.

VioCreatePS Parameter - hvps

hvps (HVIO) - input

Reserved. Must be 0.

VioCreatePS Return Value - rc

rc (APIRET) - returns
Return code.

VioCreatePS returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioCreatePS - Parameters

phvps (PHVIO) - output
Pointer to the presentation-space handle.

The location where the newly created presentation-space handle is to be returned.

Rows (ULONG) - input
Number of rows.

The number of rows in the presentation space. The maximum value allowed is 255.

Columns (ULONG) - input
Number of columns.

The number of columns in the presentation space. The maximum value allowed is 255.

Format (ULONG) - input
Format of the attributes

The attributes may be one of the following format types:

1	VGA compatible
2	Unicode
3	MFI compatible

AttrBytes (ULONG) - input
Number of attribute bytes.

This is used along with the format to select the attribute structure. This field has a value of 1, 2, or 3.

hvps (HVIO) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioCreatePS returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioCreatePS - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

VioDeleteSetId

VioDeleteSetId - Syntax

Makes the logical font no longer available through the local ID.

```
#define INCL_VIO
#include <os2.h>

ULONG     lcid;  /*  Local ID for the font. */
HVIO      hvps;  /*  VIO presentation space handle. */
APIRET    rc;    /*  Return code. */

rc = VioDeleteSetId(lcid, hvps);
```

VioDeleteSetId Parameter - lcid

lcid (ULONG) - input
Local ID for the font.

VioDeleteSetId Parameter - hvps

hvps (HVIO) - input
VIO presentation space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

VioDeleteSetId Return Value - rc

rc (APIRET) - returns
Return code.

VioDeleteSetId returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioDeleteSetId - Parameters

lcid (ULONG) - input
Local ID for the font.

hvps (HVIO) - input
VIO presentation space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioDeleteSetId returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioDeleteSetId - Remarks

After this call, the *lcid* is available for reuse.

In OS/2 2.x, *hvps* cannot be 0.

VioDeleteSetId - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioDestroyPS

VioDestroyPS - Syntax

Destroys the VIO presentation space.

```
#define INCL_VIO
#include <os2.h>

HVIO      hvps;  /*  VIO presentation-space handle. */
APIRET    rc;    /*  Return code. */

rc = VioDestroyPS(hvps);
```

VioDestroyPS Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is a value returned by VioCreatePS.

VioDestroyPS Return Value - rc

rc (APIRET) - returns
Return code.

VioDestroyPS returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
499	ERROR_VIO_ASSOCIATED_DC

VioDestroyPS - Parameters

hvps (HVIO) - input
VIO presentation-space handle.

This is a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioDestroyPS returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
499	ERROR_VIO_ASSOCIATED_DC

VioDestroyPS - Remarks

The presentation space must not be associated with a device context when VioDestroyPS is called.

The VIO presentation-space handle is invalid after this call.

After this call, the local ID for the font is available for reuse.

VioDestroyPS - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioEndPopUp

VioEndPopUp - Syntax

Redirects video output back to the normal video buffer. This function should be issued by the application when it no longer requires the temporary screen obtained through a previous VioPopUp call.

```
#define INCL_VIO
#include <os2.h>

HVIO      VioHandle;  /*  Reserved.  Must be 0. */
APIRET    rc;         /*  Return code. */

rc = VioEndPopUp(VioHandle);
```

VioEndPopUp Parameter - VioHandle

VioHandle (HVIO) - input
Reserved. Must be 0.

VioEndPopUp Return Value - rc

rc (APIRET) - returns
Return code.

VioEndPopUp returns one of the following values:

0	NO_ERROR
405	ERROR_VIO_NO_POPUP
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioEndPopUp - Parameters

VioHandle (HVIO) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioEndPopUp returns one of the following values:

0	NO_ERROR
405	ERROR_VIO_NO_POPUP
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioEndPopUp - Remarks

When the application issues a VioEndPopUp call, all video calls are directed to the application's normal video buffer. An error is returned if the call is issued with a non-zero handle.

VioEndPopUp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetAnsi

VioGetAnsi - Syntax

Returns the current ANSI status on and off state.

```
#define INCL_VIO
#include <os2.h>

PULONG    Indicator;  /*  Address of the current ANSI status. */
HVIO      VioHandle;  /*  Presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioGetAnsi(Indicator, VioHandle);
```

VioGetAnsi Parameter - Indicator

Indicator (PULONG) - output
Address of the current ANSI status.

The ANSI status may be one of the following values:

1	ANSI is active
0	ANSI is not active

VioGetAnsi Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioGetAnsi Return Value - rc

rc (APIRET) - returns
Return code.

VioGetAnsi returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetAnsi - Parameters

Indicator (PULONG) - output
Address of the current ANSI status.

The ANSI status may be one of the following values:

1	ANSI is active
0	ANSI is not active

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetAnsi returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetAnsi - Remarks

When the application issues a VioEndPopUp call, all video calls are directed to the application's normal video buffer. An error is returned if the call is issued with a non-zero handle.

VioGetAnsi - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetBuf

VioGetBuf - Syntax

Returns the address of the logical video buffer (LVB).

```
#define INCL_VIO
#include <os2.h>

PULONG    LVBPtr;     /*  Pointer to the logical video buffer address. */
PULONG    Length;     /*  Pointer to the length of the buffer, in bytes. */
HVIO      VioHandle;  /*  Presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioGetBuf(LVBPtr, Length, VioHandle);
```

VioGetBuf Parameter - LVBPtr

LVBPtr (PULONG) - output
Pointer to the logical video buffer address.

VioGetBuf Parameter - Length

Length (PULONG) - output
Pointer to the length of the buffer, in bytes.

The length is the number of rows, times the number of columns, times the size of the cell.

VioGetBuf Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioGetBuf Return Value - rc

rc (APIRET) - returns
Return code.

VioGetBuf returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioGetBuf - Parameters

LVBPtr (PULONG) - output
Pointer to the logical video buffer address.

Length (PULONG) - output
Pointer to the length of the buffer, in bytes.

The length is the number of rows, times the number of columns, times the size of the cell.

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetBuf returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioGetBuf - Remarks

An application using VioGetBuf can prepare a screen in the application's own logical video buffer (LVB) offline. When the application is in the foreground, the physical screen buffer is updated from the LVB when VioShowBuf is issued. When the application runs in the background, the physical screen buffer is updated when the application is switched to the foreground.

Once VioGetBuf is issued, all VioWrtXX calls issued while the application is running in the foreground are written to the physical display buffer and LVB.

VioGetMode can be used to determine the dimensions of the buffer.

If VioSetMode is issued following a VioGetBuf call, the size of the LVB is changed, and VioGetBuf must be reissued.

VioGetBuf - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetConfig

VioGetConfig - Syntax

Returns the video display configuration.

```
#define INCL_VIO
#include <os2.h>

ULONG             ConfigID;    /*  Configuration ID. */
PVIOCONFIGINFO    ConfigData;  /*  Pointer to the configuration data. */
HVIO              VioHandle;   /*  Presentation-space handle. */
APIRET            rc;          /*  Return code. */

rc = VioGetConfig(ConfigID, ConfigData, VioHandle);
```

VioGetConfig Parameter - ConfigID

ConfigID (ULONG) - input
Configuration ID.

Identifies the display configuration for which information is being requested:

Value	Definition
0	Current configuration
1	Primary configuration
2	Secondary configuration

VioGetConfig Parameter - ConfigData

ConfigData (PVIOCONFIGINFO) - output
Pointer to the configuration data.

VioGetConfig Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioGetConfig Return Value - rc

rc (APIRET) - returns
Return code.

VioGetConfig returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioGetConfig - Parameters

ConfigID (ULONG) - input
Configuration ID.

Identifies the display configuration for which information is being requested:

Value	Definition
0	Current configuration
1	Primary configuration
2	Secondary configuration

ConfigData (PVIOCONFIGINFO) - output
Pointer to the configuration data.

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetConfig returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioGetConfig - Remarks

The values returned might not be correct if the adapter cannot be properly identified or if the device is not capable of returning its settings.

VioGetConfig - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetCp

VioGetCp - Syntax

Queries the code page currently used to display text data.

```
#define INCL_VIO
#include <os2.h>

ULONG      Reserved;     /*  Reserved.  Must be 0. */
PUSHORT    CodePageID;   /*  Code-page ID. */
HVIO       VioHandle;    /*  Presentation-space handle. */
APIRET     rc;           /*  Return code. */
```

```
rc = VioGetCp(Reserved, CodePageID, VioHandle);
```

VioGetCp Parameter - Reserved

Reserved (ULONG) - input
Reserved. Must be 0.

VioGetCp Parameter - CodePageID

CodePageID (PUSHORT) - output
Code-page ID.

The address of a word in the application's data area. The current video code page is returned in this word.

VioGetCp Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioGetCp Return Value - rc

rc (APIRET) - returns
Return code.

VioGetCp returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetCp - Parameters

Reserved (ULONG) - input
Reserved. Must be 0.

CodePageID (PUSHORT) - output
Code-page ID.

The address of a word in the application's data area. The current video code page is returned in this word.

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetCp returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetCp - Remarks

The display code-page ID, previously set by VioSetCp or inherited from the requesting process, is returned to the caller. The returned code-page tag is the currently active code page.

VioGetCp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetCurPos

VioGetCurPos - Syntax

Returns the coordinates of the cursor.

```
#define INCL_VIO
#include <os2.h>

PULONG     Row;         /*  Row return data. */
PUSHORT    Column;      /*  Column return data. */
HVIO       VioHandle;   /*  Presentation-space handle. */
APIRET     rc;          /*  Return code. */

rc = VioGetCurPos(Row, Column, VioHandle);
```

VioGetCurPos Parameter - Row

Row (PULONG) - output
Row return data.

Address of the current row position of the cursor, where 0 is the top row.

VioGetCurPos Parameter - Column

Column (PUSHORT) - output
Column return data.

The address of the current column position of the cursor, where 0 is the leftmost column.

VioGetCurPos Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be zero, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS

VioGetCurPos Return Value - rc

rc (APIRET) - returns
Return code.

VioGetCurPos returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetCurPos - Parameters

Row (PULONG) - output
Row return data.

Address of the current row position of the cursor, where 0 is the top row.

Column (PUSHORT) - output

Column return data.

The address of the current column position of the cursor, where 0 is the leftmost column.

VioHandle (HVIO) - input
Presentation-space handle.

This must be zero, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS

rc (APIRET) - returns
Return code.

VioGetCurPos returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetCurPos - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

VioGetCurType

VioGetCurType - Syntax

Returns the cursor type.

```
#define INCL_VIO
#include <os2.h>

PVIOCURSORINFO    CursorData;  /*  Cursor characteristics. */
HVIO              VioHandle;   /*  Presentation-space handle. */
APIRET            rc;          /*  Return code. */

rc = VioGetCurType(CursorData, VioHandle);
```

VioGetCurType Parameter - CursorData

CursorData (PVIOCURSORINFO) - output
Cursor characteristics.

Address of the cursor-characteristics structure.

VioGetCurType Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS

VioGetCurType Return Value - rc

rc (APIRET) - returns
Return code.

VioGetCurType returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetCurType - Parameters

CursorData (PVIOCURSORINFO) - output
Cursor characteristics.

Address of the cursor-characteristics structure.

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS

rc (APIRET) - returns
Return code.

VioGetCurType returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetCurType - Remarks

The cursor start line (*yStart*) and cursor end line (*cEnd*) were originally specified as percentages on VioSetCurType (using negative values); the positive values, into which they were translated, are returned. See VioSetCurType for more information on how percentages can be used to set *yStart* and *cEnd* independent of the number of scan lines per character cell.

VioGetCurType - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetDeviceCellSize

VioGetDeviceCellSize - Syntax

Returns the size of the current character cell in pels.

```
#define INCL_VIO
#include <os2.h>

PULONG    Height;
PULONG    Width;
HVIO      hvps;     /*  VIO presentation-space handle. */
APIRET    rc;       /*  Return code. */

rc = VioGetDeviceCellSize(Height, Width, hvps);
```

VioGetDeviceCellSize Parameter - Height

Height (PULONG) - input
Pointer to the height of the character cell in pels.

VioGetDeviceCellSize Parameter - Width

Width (PULONG) - input
Pointer to the width of the character cell in pels.

VioGetDeviceCellSize Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is a value returned by VioCreatePS.

VioGetDeviceCellSize Return Value - rc

rc (APIRET) - returns
Return code.

VioGetDeviceCellSize returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetDeviceCellSize - Parameters

Height (PULONG) - input
Pointer to the height of the character cell in pels.

Width (PULONG) - input
Pointer to the width of the character cell in pels.

hvps (HVIO) - input
VIO presentation-space handle.

This is a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetDeviceCellSize returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetDeviceCellSize - Remarks

In OS/2 2.x, *hvps* cannot be 0.

VioGetDeviceCellSize - Topics

Select an item:
Syntax
Parameters

Returns
Remarks
Glossary

VioGetMode

VioGetMode - Syntax

Returns the mode of the display.

```
#define INCL_VIO
#include <os2.h>

PVIOMODEINFO    ModeData;   /*  Mode characteristics. */
HVIO            VioHandle;  /*  VIO presentation-space handle. */
APIRET          rc;         /*  Return code. */

rc = VioGetMode(ModeData, VioHandle);
```

VioGetMode Parameter - ModeData

ModeData (PVIOMODEINFO) - in/out
Mode characteristics.

The address of a structure where mode characteristics are returned.

VioGetMode Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioGetMode Return Value - rc

rc (APIRET) - returns
Return code.

VioGetMode returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioGetMode - Parameters

ModeData (PVIOMODEINFO) - in/out
Mode characteristics.

The address of a structure where mode characteristics are returned.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetMode returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioGetMode - Remarks

See VioSetMode for examples.

VioGetMode - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetOrigin

VioGetOrigin - Syntax

Gets the position at which the presentation space maps to the window.

```
#define INCL_VIO
#include <os2.h>

PULONG    Row;
PULONG    Column;
HVIO      hvps;    /*  VIO presentation-space handle. */
APIRET    rc;      /*  Return code. */

rc = VioGetOrigin(Row, Column, hvps);
```

VioGetOrigin Parameter - Row

Row (PULONG) - input
Location to return the top-most row shown in the window.

VioGetOrigin Parameter - Column

Column (PULONG) - input
Location to return the left-most column shown in the window.

VioGetOrigin Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

VioGetOrigin Return Value - rc

rc (APIRET) - returns
Return code.

VioGetOrigin returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetOrigin - Parameters

Row (PULONG) - input
Location to return the top-most row shown in the window.

Column (PULONG) - input
Location to return the left-most column shown in the window.

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioGetOrigin returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioGetOrigin - Remarks

In OS/2 2.x, *hvps* cannot be 0.

VioGetOrigin - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioGetState

VioGetState - Syntax

Gets the current settings of the:

- Blink and background intensity switch
- Color registers
- Overscan (border) color
- Palette registers
- Target VioSetMode display configuration
- Underline location

```
#define INCL_VIO
#include <os2.h>

PVIOD     RequestBlock;  /*  Request block. */
HVIO      VioHandle;     /*  Reserved.  Must be 0. */
APIRET    rc;            /*  Return code. */

rc = VioGetState(RequestBlock, VioHandle);
```

VioGetState Parameter - RequestBlock

RequestBlock (PVIOD) - in/out
Request block.

The address of the video-state structures consisting of six different structures depending on the request type:

Type	Definition
0	Get palette registers
1	Get overscan (border) color
2	Get blink and background intensity switch
3	Get color registers
4	Reserved
5	Get the scan line for underlining
6	Get target VioSetMode display configuration

The six structures, depending on request type, are:

- VIOPALSTATE
- VIOOVERSCAN
- VIOINTENSITY
- VIOCOLORREG
- VIOSETULINELOC
- VIOSETTARGET

VioGetState Parameter - VioHandle

VioHandle (HVIO) - input
Reserved. Must be 0.

VioGetState Return Value - rc

rc (APIRET) - returns
Return code.

VioGetState returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioGetState - Parameters

RequestBlock (PVIOD) - in/out
Request block.

The address of the video-state structures consisting of six different structures depending on the request type:

Type	Definition
0	Get palette registers
1	Get overscan (border) color
2	Get blink and background intensity switch
3	Get color registers
4	Reserved
5	Get the scan line for underlining
6	Get target VioSetMode display configuration

The six structures, depending on request type, are:

- VIOPALSTATE
- VIOOVERSCAN
- VIOINTENSITY
- VIOCOLORREG
- VIOSETULINELOC
- VIOSETTARGET

VioHandle (HVIO) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioGetState returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioGetState - Remarks

Note: VioGetState allows access to hardware-dependent features. Not all video hardware will honor these settings or return valid settings.

VioGetState - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioModeUndo

VioModeUndo - Syntax

Allows one thread within a process to cancel a VioModeWait issued by another thread within the same process.

```
#define INCL_VIO
#include <os2.h>

ULONG     OwnerIndic;  /*  Ownership indicator. */
ULONG     KillIndic;   /*  Terminate indicator */
ULONG     Reserved;    /*  Reserved.  Must be 0. */
APIRET    rc;          /*  Return code. */

rc = VioModeUndo(OwnerIndic, KillIndic, Reserved);
```

VioModeUndo Parameter - OwnerIndic

OwnerIndic (ULONG) - input
Ownership indicator.

Indicates whether the thread issuing VioModeUndo wants ownership of VioModeWait to be reserved for its process.

Value	Definition
0	Reserve ownership
1	Give up ownership.

VioModeUndo Parameter - KillIndic

KillIndic (ULONG) - input
Terminate indicator

Indicates whether the thread (with the outstanding VioModeWait) should return an error code or be terminated.

Value	Definition
0	Return error code.
1	Terminate thread.

VioModeUndo Parameter - Reserved

Reserved (ULONG) - input
Reserved. Must be 0.

VioModeUndo Return Value - rc

rc (APIRET) - returns
Return code.

VioModeUndo returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
427	ERROR_VIO_NO_MODE_THREAD
430	ERROR_VIO_ILLEGAL_DURING_POPUP
486	ERROR_VIO_BAD_RESERVE

VioModeUndo - Parameters

OwnerIndic (ULONG) - input
Ownership indicator.

Indicates whether the thread issuing VioModeUndo wants ownership of VioModeWait to be reserved for its process.

Value	Definition
0	Reserve ownership
1	Give up ownership.

KillIndic (ULONG) - input
Terminate indicator

Indicates whether the thread (with the outstanding VioModeWait) should return an error code or be terminated.

Value	Definition
0	Return error code.
1	Terminate thread.

Reserved (ULONG) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioModeUndo returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
427	ERROR_VIO_NO_MODE_THREAD
430	ERROR_VIO_ILLEGAL_DURING_POPUP
486	ERROR_VIO_BAD_RESERVE

VioModeUndo - Remarks

VioModeUndo can be issued only by a thread within the process that owns VioModeWait. The thread issuing VioModeUndo can either reserve ownership of the VioModeWait call for its process or give up ownership. The thread whose VioModeWait is canceled is optionally terminated.

VioModeUndo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioModeWait

VioModeWait - Syntax

Notifies a graphics-mode application when it must restore its video mode, state, and modified display-adapter registers. The return from this call provides the notification.

```
#define INCL_VIO
#include <os2.h>

ULONG     RequestType;  /*  Request type. */
PULONG    NotifyType;   /*  Notify type. */
ULONG     Reserved;     /*  Reserved.  Must be 0. */
APIRET    rc;           /*  Return code. */

rc = VioModeWait(RequestType, NotifyType,
       Reserved);
```

VioModeWait Parameter - RequestType

RequestType (ULONG) - input
Request type.

Valid request types are:

0 Indicates the application wants to be notified at the end of a pop-up to restore its mode.

VioModeWait Parameter - NotifyType

NotifyType (PULONG) - output
Notify type.

Address of the of the operation to be performed by the application returning from VioModeWait. Valid notify types are:

0 Restore mode.

VioModeWait Parameter - Reserved

Reserved (ULONG) - input
Reserved. Must be 0.

VioModeWait Return Value - rc

rc (APIRET) - returns
Return code.

VioModeWait returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
423	ERROR_VIO_RETURN
424	ERROR_SCS_INVALID_FUNCTION
428	ERROR_VIO_NO_SAVE_RESTORE_THD
430	ERROR_VIO_ILLEGAL_DURING_POPUP

VioModeWait - Parameters

RequestType (ULONG) - input
Request type.

Valid request types are:

0	Indicates the application wants to be notified at the end of a pop-up to restore its mode.

NotifyType (PULONG) - output
Notify type.

Address of the of the operation to be performed by the application returning from VioModeWait. Valid notify types are:

0	Restore mode.

Reserved (ULONG) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioModeWait returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
423	ERROR_VIO_RETURN
424	ERROR_SCS_INVALID_FUNCTION
428	ERROR_VIO_NO_SAVE_RESTORE_THD
430	ERROR_VIO_ILLEGAL_DURING_POPUP

VioModeWait - Remarks

At the completion of an application or hard-error pop-up (see VioPopUp), the OS/2 operating system notifies the session that was originally interrupted for the pop-up to restore its mode. The return from VioModeWait provides that notification. The thread that issued the call must perform the restore and then immediately reissue VioModeWait.

When an application's VioModeWait thread is notified, the thread must restore its video mode, state, and modified display-adapter registers. An application's VioModeWait thread does not restore the physical display buffer. The OS/2 operating system saves and/or restores the physical display buffer over a pop-up.

Only one process for a session can issue VioModeWait. The first process that issues VioModeWait becomes the owner of this function. (See VioModeUndo.)

An application must issue VioModeWait only if it writes directly to the registers on the display adapter. Otherwise, the application can allow the OS/2 operating system to perform the required restoration by not issuing VioModeWait.

When an application issues VioModeWait, it is also required to issue VioSavRedrawWait to be notified at screen switch time to perform a full save or restoration (see VioSavRedrawWait). Two application threads must be dedicated to performing these operations.

VioModeWait - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioPopUp

VioPopUp - Syntax

Displays a temporary screen with a message to the user.

```
#define INCL_VIO
#include <os2.h>

PULONG    Options;    /*  Option flags. */
HVIO      VioHandle;  /*  Presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioPopUp(Options, VioHandle);
```

VioPopUp Parameter - Options

Options (PULONG) - input
Option flags.

Bit	Description
31-2	Reserved. Set to 0.
1	0 = Non-transparent operation. The video mode is set to a text mode. The screen is cleared, and the cursor is positioned at the upper-left corner of the screen. 1 = Transparent operation. If the video mode of the outgoing foreground session is a text mode, no mode change occurs. The screen is not cleared, and the cursor remains at its current position. If the video mode is not text, or if this session has a VioSavRedrawWait active, the pop-up is refused. The OS/2 operating system is responsible for saving and restoring the physical display buffer of the previous foreground session around a pop-up. This is true whether transparent or nontransparent operation is selected.
0	0 = Return with unique error code, ERROR_VIO_EXISTING_POPUP, if pop-up is not immediately available. 1 = Wait if pop-up is not immediately available.

VioPopUp Parameter - VioHandle

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioPopUp Return Value - rc

rc (APIRET) - returns
Return code.

VioPopUp returns one of the following values:

0	NO_ERROR
405	ERROR_VIO_NO_POPUP
406	ERROR_VIO_EXISTING_POPUP
421	ERROR_VIO_INVALID_PARMS
483	ERROR_VIO_TRANSPARENT_POPUP

VioPopUp - Parameters

Options (PULONG) - input
Option flags.

Bit	Description
31-2	Reserved. Set to 0.
1	0 = Non-transparent operation. The video mode is set to a text mode. The screen is cleared, and the cursor is positioned at the upper-left corner of the screen. 1 = Transparent operation. If the video mode of the outgoing foreground session is a text mode, no

mode change occurs. The screen is not cleared, and the cursor remains at its current position. If the video mode is not text, or if this session has a VioSavRedrawWait active, the pop-up is refused.

The OS/2 operating system is responsible for saving and restoring the physical display buffer of the previous foreground session around a pop-up. This is true whether transparent or nontransparent operation is selected.

0 — 0 = Return with unique error code, ERROR_VIO_EXISTING_POPUP, if pop-up is not immediately available.

1 = Wait if pop-up is not immediately available.

VioHandle (HVIO) - input
Presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioPopUp returns one of the following values:

0	NO_ERROR
405	ERROR_VIO_NO_POPUP
406	ERROR_VIO_EXISTING_POPUP
421	ERROR_VIO_INVALID_PARMS
483	ERROR_VIO_TRANSPARENT_POPUP

VioPopUp - Remarks

VioPopUp is normally issued by the application when it is running in the background and wishes to immediately display a message to the user without waiting to become the active foreground session.

When an application process issues VioPopUp, it should wait for the return from the request. If the process allows any of its threads to write to the screen before VioPopUp returns a successful return code, the screen output might be directed to the application's normal video buffer rather than to the pop-up screen. If the process allows any of its threads to issue keyboard or mouse calls before VioPopUp returns a successful return code, the input is directed from the application's normal session. Once the process that issued VioPopUp receives a successful return code, video and keyboard calls issued by any of the threads in the pop-up process are directed to the pop-up screen. This continues until the process issues VioEndPopUp. At that time, video and keyboard calls resume being directed to the application's normal video buffer.

Only one pop-up can exist at any time. If a process requests a pop-up and a pop-up already exists, the process can either wait for the prior pop-up to end or receive an immediate return with an error code. The error code indicates that the operation failed due to an existing pop-up having captured the screen.

Video pop-ups provide a mechanism for a background application to notify the operator of an abnormal event that requires the operator to take some action. When considering the suitability of using pop-ups in a particular situation, the possible disruptive effect of pop-ups to the operator should be considered. If the operator were interrupted frequently by pop-ups issued by background applications, the operator would not work effectively with the foreground application.

While a video pop-up is in the foreground, the operator cannot use the hot key to switch to another application or to the shell. Before the operator can switch to another application, or switch the shell to the foreground, the pop-up application must issue VioEndPopUp.

While a video pop-up is in effect, all video calls from the previous foreground session are blocked until the process that issued VioPopUp issues VioEndPopUp.

When VioPopUp is issued, only the process within the session that issued VioPopUp is brought to the foreground. Assuming the session was already the foreground session, any video calls issued by other processes in that session are blocked until the process that issued VioPopUp issues VioEndPopUp. VioEndPopUp.

DosExecPgm cannot be issued by a process during a pop-up. The following video calls are the only calls that can be issued during the pop-up by the process that issued VioPopUp:

VioEndPopUp	VioScrollLeft
VioGetConfig	VioSetCurPos
VioGetCp	VioSetCurType
VioGetAnsi	VioSetCp

VioGetState
VioGetCurPos
VioGetCurType
VioGetMode
VioReadCellStr
VioReadCharStr
VioScrollRight
VioScrollUp
VioScrollDown
VioSetState
VioWrtNChar
VioWrtNAttr
VioWrtNCell
VioWrtCharStr
VioWrtCharStrAtt
VioWrtCellStr
VioWrtTTY

This function can be used from within a PM application. Kbdxxx, Mouxxx, and Vioxxx calls with a zero handle are all allowed between VioPopUp and VioEndPopUp and are directed to the pop-up screen.

VioPopUp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioQueryFonts

VioQueryFonts - Syntax

Gets the position at which the presentation space maps to the window.

```
#define INCL_VIO
#include <os2.h>

PULONG          Remfonts;  /*  Number of fonts not returned. */
PFONTMETRICS    Metrics;   /*  Font metrics. */
ULONG           Metlen;    /*  Maximum length of data. */
PULONG          Fonts;     /*  Number of fonts. */
PSZ             Facename;  /*  Face name of fonts. */
ULONG           Options;   /*  Controls which fonts are selected. */
HVIO            hvps;      /*  VIO presentation-space handle. */
APIRET          rc;        /*  Return code. */

rc = VioQueryFonts(Remfonts, Metrics, Metlen,
       Fonts, Facename, Options, hvps);
```

VioQueryFonts Parameter - Remfonts

Remfonts (PULONG) - output
Number of fonts not returned.

The number of fonts for which information is not returned.

VioQueryFonts Parameter - Metrics

Metrics (PFONTMETRICS) - output
Font metrics.

Matching font metrics are returned in this buffer.

VioQueryFonts Parameter - Metlen

Metlen (ULONG) - input
Maximum length of data.

The maximum length of data to be returned for each record.

VioQueryFonts Parameter - Fonts

Fonts (PULONG) - in/out
Number of fonts.

The location of the number of fonts to be returned on input. This is updated with the actual number of fonts returned on output.

VioQueryFonts Parameter - Facename

Facename (PSZ) - input
Face name of fonts.

The face name of fonts desired, or NULL to indicate that all applicable fonts should be returned.

VioQueryFonts Parameter - Options

Options (ULONG) - input
Controls which fonts are selected.

Valid values are:

VQF_PUBLIC	Return only public fonts.
VQF_PRIVATE	Return only private fonts.
VQF_ALL	Return both public and private fonts.

VioQueryFonts Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

VioQueryFonts Return Value - rc

rc (APIRET) - returns
Return code.

VioQueryFonts returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioQueryFonts - Parameters

Remfonts (PULONG) - output
Number of fonts not returned.

The number of fonts for which information is not returned.

Metrics (PFONTMETRICS) - output
Font metrics.

Matching font metrics are returned in this buffer.

Metlen (ULONG) - input
Maximum length of data.

The maximum length of data to be returned for each record.

Fonts (PULONG) - in/out
Number of fonts.

The location of the number of fonts to be returned on input. This is updated with the actual number of fonts returned on output.

Facename (PSZ) - input
Face name of fonts.

The face name of fonts desired, or NULL to indicate that all applicable fonts should be returned.

Options (ULONG) - input
Controls which fonts are selected.

Valid values are:

VQF_PUBLIC	Return only public fonts.
VQF_PRIVATE	Return only private fonts.
VQF_ALL	Return both public and private fonts.

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioQueryFonts returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioQueryFonts - Remarks

By inspecting the returned font metrics, the application can choose the font that best meets its requirements.

All metrics are returned in pel coordinates.

In OS/2 2.x, *hvps* cannot be zero.

VioQueryFonts - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioQuerySetIds

VioQuerySetIds - Syntax

Gets information about VIO local identifiers.

```
#define INCL_VIO
#include <os2.h>

PULONG    lcids;  /*  Array of local identifiers. */
PSTR8     Names;
PULONG    Types;
PULONG    count;  /*  Number of objects to be queried. */
```

```
HVIO      hvps;    /*  VIO presentation-space handle. */
APIRET    rc;      /*  Return code. */

rc = VioQuerySetIds(lcids, Names, Types, count,
       hvps);
```

VioQuerySetIds Parameter - lcids

lcids (PULONG) - output
Array of local identifiers.

VioQuerySetIds Parameter - Names

Names (PSTR8) - output
An array of 8 character names associated with the *lcids* .

VioQuerySetIds Parameter - Types

Types (PULONG) - output
An array of types associated with each *lcids* .

VioQuerySetIds Parameter - count

count (PULONG) - input
Number of objects to be queried.

The maximum value in use is 3.

VioQuerySetIds Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

VioQuerySetIds Return Value - rc

rc (APIRET) - returns
Return code.

VioQuerySetIds returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioQuerySetIds - Parameters

lcids (PULONG) - output
Array of local identifiers.

Names (PSTR8) - output
An array of 8 character names associated with the *lcids* .

Types (PULONG) - output
An array of types associated with each *lcids* .

count (PULONG) - input
Number of objects to be queried.

The maximum value in use is 3.

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session, or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioQuerySetIds returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioQuerySetIds - Remarks

In OS/2 2.x, *hvps* cannot be 0.

VioQuerySetIds - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioReadCellStr

VioReadCellStr - Syntax

Reads a string of character-attribute pairs (cells) from the screen, starting at the specified location.

```
#define INCL_VIO
#include <os2.h>

PCH       CellStr;    /*  Cell string buffer. */
PULONG    Length;     /*  Length of cell string buffer. */
ULONG     Row;        /*  Starting row location. */
ULONG     Column;     /*  Starting column location. */
HVIO      VioHandle;  /*  Vio presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioReadCellStr(CellStr, Length, Row,
       Column, VioHandle);
```

VioReadCellStr Parameter - CellStr

CellStr (PCH) - output
Cell string buffer.

Address of the buffer where the cell string is returned.

VioReadCellStr Parameter - Length

Length (PULONG) - in/out
Length of cell string buffer.

Address of the buffer length in bytes. Length must take into account that each character-attribute entry in the buffer is 2 or 4 bytes. If the length of the buffer is not sufficient, the last entry is not complete.

VioReadCellStr Parameter - Row

Row (ULONG) - input
Starting row location.

Starting row of the field to read, where 0 is the top row.

VioReadCellStr Parameter - Column

Column (ULONG) - input
Starting column location.

Starting column of the field to read, where 0 is the leftmost column.

VioReadCellStr Parameter - VioHandle

VioHandle (HVIO) - input
Vio presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case it must be the value returned by VioCreatePS.

VioReadCellStr Return Value - rc

rc (APIRET) - returns
Return code.

VioReadCellStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioReadCellStr - Parameters

CellStr (PCH) - output
Cell string buffer.

Address of the buffer where the cell string is returned.

Length (PULONG) - in/out
Length of cell string buffer.

Address of the buffer length in bytes. Length must take into account that each character-attribute entry in the buffer is 2 or 4 bytes. If the length of the buffer is not sufficient, the last entry is not complete.

Row (ULONG) - input
Starting row location.

Starting row of the field to read, where 0 is the top row.

Column (ULONG) - input
Starting column location.

Starting column of the field to read, where 0 is the leftmost column.

VioHandle (HVIO) - input
Vio presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioReadCellStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioReadCellStr - Remarks

If a string read comes to the end of the line and is not complete, it continues at the beginning of the next line. If the read comes to the end of the screen and is not complete, it terminates and the length is set to the length of the buffer that was filled.

VioReadCellStr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioReadCharStr

VioReadCharStr - Syntax

Reads a string of characters from the display, starting at the specified location.

```
#define INCL_VIO
#include <os2.h>

PCH       CharStr;    /*  Character buffer. */
PULONG    Length;     /*  Length of buffer. */
ULONG     Row;        /*  Starting row location. */
ULONG     Column;     /*  Starting column location. */
HVIO      VioHandle;  /*  Vio presentation-space handle. */
APIRET    rc;         /*  Return code. */
```

```
rc = VioReadCharStr(CharStr, Length, Row,
       Column, VioHandle);
```

VioReadCharStr Parameter - CharStr

CharStr (PCH) - output
Character buffer.

Address of the buffer where the character string is returned.

VioReadCharStr Parameter - Length

Length (PULONG) - in/out
Length of buffer.

Address of the buffer length in bytes.

VioReadCharStr Parameter - Row

Row (ULONG) - input
Starting row location.

Starting row of the field to read, where 0 is the top row.

VioReadCharStr Parameter - Column

Column (ULONG) - input
Starting column location.

Starting column of the field to read, where 0 is the leftmost column.

VioReadCharStr Parameter - VioHandle

VioHandle (HVIO) - input
Vio presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioReadCharStr Return Value - rc

rc (APIRET) - returns
Return code.

VioReadCharStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioReadCharStr - Parameters

CharStr (PCH) - output
Character buffer.

Address of the buffer where the character string is returned.

Length (PULONG) - in/out
Length of buffer.

Address of the buffer length in bytes.

Row (ULONG) - input
Starting row location.

Starting row of the field to read, where 0 is the top row.

Column (ULONG) - input
Starting column location.

Starting column of the field to read, where 0 is the leftmost column.

VioHandle (HVIO) - input
Vio presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioReadCharStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioReadCharStr - Remarks

If a string read comes to the end of the line and is not complete, it continues at the beginning of the next line. If the read comes to the end of

the screen and is not complete, it terminates, and the length is set to the number of characters read.

VioReadCharStr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSavRedrawWait

VioSavRedrawWait - Syntax

Notifies a graphics mode application when it must save or redraw its screen image.

```
#define INCL_VIO
#include <os2.h>

ULONG     SaveRedrawIndic;  /*  Save/redraw indicator. */
PULONG    NotifyType;       /*  Notify type (returned). */
HVIO      VioHandle;        /*  Reserved.  Must be 0. */
APIRET    rc;               /*  Return code. */

rc = VioSavRedrawWait(SaveRedrawIndic, NotifyType,
       VioHandle);
```

VioSavRedrawWait Parameter - SaveRedrawIndic

SaveRedrawIndic (ULONG) - input
Save/redraw indicator.

Indicates which events the application is waiting for:

Value	Definition
0	The session manager notifies the application for both save and redraw operations.
1	The session manager notifies the application for redraw operations only.

VioSavRedrawWait Parameter - NotifyType

NotifyType (PULONG) - output

Notify type (returned).

Address that specifies the operation to be performed by the application upon return from VioSavRedrawWait:

Value	Definition
0	Save screen image.
1	Restore screen image.

VioSavRedrawWait Parameter - VioHandle

VioHandle (HVIO) - input
Reserved. Must be 0.

VioSavRedrawWait Return Value - rc

rc (APIRET) - returns
Return code.

VioSavRedrawWait returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
423	ERROR_VIO_RETURN
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioSavRedrawWait - Parameters

SaveRedrawIndic (ULONG) - input
Save/redraw indicator.

Indicates which events the application is waiting for:

Value	Definition
0	The session manager notifies the application for both save and redraw operations.
1	The session manager notifies the application for redraw operations only.

NotifyType (PULONG) - output
Notify type (returned).

Address that specifies the operation to be performed by the application upon return from VioSavRedrawWait:

Value	Definition
0	Save screen image.
1	Restore screen image.

VioHandle (HVIO) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioSavRedrawWait returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
423	ERROR_VIO_RETURN
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioSavRedrawWait - Remarks

The OS/2 operating system uses VioSavRedrawWait to notify a graphics-mode application to save or restore its screen image at screen switch time. The application in the outgoing foreground session is notified to perform a save. The application in the incoming foreground session is notified to perform a restore. The application must perform the action requested and immediately reissue VioSavRedrawWait. When an application performs a save, it saves its physical display buffer, video mode, and any other information the application needs to completely redraw its screen at restore time.

Only one process per session can issue VioSavRedrawWait. The process that first issues it becomes the owner of the function.

A text-mode application must issue VioSavRedrawWait only if the application writes directly to the registers on the display adapter. Assuming VioSavRedrawWait is not issued by a text-mode application, the OS/2 operating system performs the required saves and restores.

An application that issues VioSavRedrawWait might also need to issue VioModeWait. This would allow the application to be notified when it must restore its mode at the completion of an application or hard-error pop-up. See VioModeWait for more information. Two application threads would be required to perform these operations, in this case.

At the time a VioSavRedrawWait thread is notified, the session is in transition to or from the background. Although the session's official status is background, any selector to the physical display buffer previously obtained by the VioSavRedrawWait process (through VioGetPhysBuf) is valid, at this time. The physical display buffer must be accessed without issuing VioScrLock. Because the session's official status is background, any thread waits if it issues VioScrLock with the "wait if unsuccessful" option.

An application containing a VioSavRedrawWait thread should be designed so that the process does not cause any hard errors while the VioSavRedrawWait thread is running; otherwise, a system lockout might occur.

An application's VioSavRedrawWait thread might be notified to perform a restore before it is notified to perform a save. This happens if the application was running in the background the first time it issued VioSavRedrawWait. The return from this function call provides the notification. The thread that issues the call performs the save or redraw and then reissues VioSavRedrawWait to wait until its screen image must be saved or redrawn again.

VioSavRedrawWait - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSavRedrawUndo

VioSavRedrawUndo - Syntax

Allows one thread within a process to cancel a VioSavRedrawWait issued by another thread within the same process.

```
#define INCL_VIO
#include <os2.h>

ULONG     OwnerIndic;  /*  Ownership indicator */
ULONG     KillIndic;   /*  Terminate indicator */
HVIO      VioHandle;   /*  Reserved.  Must be 0. */
APIRET    rc;          /*  Return code. */

rc = VioSavRedrawUndo(OwnerIndic, KillIndic,
       VioHandle);
```

VioSavRedrawUndo Parameter - OwnerIndic

OwnerIndic (ULONG) - input
Ownership indicator

Indicates whether the thread issuing VioSavRedrawUndo wants ownership of VioSavRedrawUndo to be reserved for its process.

Value	Definition
0	Reserve ownership.
1	Give up ownership.

VioSavRedrawUndo Parameter - KillIndic

KillIndic (ULONG) - input
Terminate indicator

Indicates whether the thread (with the outstanding VioSavRedrawUndo) should return an error code or be terminated.

Value	Definition
0	Return error code.
1	Terminate thread.

VioSavRedrawUndo Parameter - VioHandle

VioHandle (HVIO) - input
Reserved. Must be 0.

VioSavRedrawUndo Return Value - rc

rc (APIRET) - returns
Return code.

VioSavRedrawUndo returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
428	ERROR_VIO_NO_SAVE_RESTORE_THD
430	ERROR_VIO_ILLEGAL_DURING_POPUP

VioSavRedrawUndo - Parameters

OwnerIndic (ULONG) - input
Ownership indicator

Indicates whether the thread issuing VioSavRedrawUndo wants ownership of VioSavRedrawUndo to be reserved for its process.

Value	Definition
0	Reserve ownership.
1	Give up ownership.

KillIndic (ULONG) - input
Terminate indicator

Indicates whether the thread (with the outstanding VioSavRedrawUndo) should return an error code or be terminated.

Value	Definition
0	Return error code.
1	Terminate thread.

VioHandle (HVIO) - input
Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioSavRedrawUndo returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
428	ERROR_VIO_NO_SAVE_RESTORE_THD
430	ERROR_VIO_ILLEGAL_DURING_POPUP

VioSavRedrawUndo - Remarks

The issuing thread can reserve ownership of VioSavRedrawWait for its process or give it up. The thread whose VioSavRedrawWait was canceled is optionally terminated. VioSavRedrawUndo can be issued only by a thread within the same process that owns VioSavRedrawWait. VioSavRedrawWait.

VioSavRedrawUndo - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioScrLock

VioScrLock - Syntax

Requests ownership of (locks) the physical display buffer.

```
#define INCL_VIO
#include <os2.h>

ULONG     WaitFlag;   /*  Wait until screen I/O can take place. */
PUCHAR    Status;     /*  Address of the lock status. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioScrLock(WaitFlag, Status, VioHandle);
```

VioScrLock Parameter - WaitFlag

WaitFlag (ULONG) - input
Wait until screen I/O can take place.

Indicates whether the process should block until the screen I/O can take place.

Value	Definition
0	Return if screen I/O not available.
1	Wait until screen I/O is available.

VioScrLock Parameter - Status

Status (PUCHAR) - output
Address of the lock status.

The lock status may be one of the following values:

Value	Definition
0	Lock successful.
1	Lock unsuccessful (in the case of no wait). Status is returned only when AX = 0. Status = 1 can be returned only when WaitFlag = 0.

VioScrLock Parameter - VioHandle

VioHandle (HVIO) - input

VIO presentation-space handle.

Reserved. Must be 0.

VioScrLock Return Value - rc

rc (APIRET) - returns
Return code.

VioScrLock returns one of the following values:

0	NO_ERROR
366	ERROR_VIO_WAIT_FLAG
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
434	ERROR_VIO_LOCK
436	ERROR_VIO_INVALID_HANDLE

VioScrLock - Parameters

WaitFlag (ULONG) - input
Wait until screen I/O can take place.

Indicates whether the process should block until the screen I/O can take place.

Value	Definition
0	Return if screen I/O not available.
1	Wait until screen I/O is available.

Status (PUCHAR) - output
Address of the lock status.

The lock status may be one of the following values:

Value	Definition
0	Lock successful.
1	Lock unsuccessful (in the case of no wait). Status is returned only when AX = 0. Status = 1 can be returned only when WaitFlag = 0.

VioHandle (HVIO) - input
VIO presentation-space handle.

Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioScrLock returns one of the following values:

0	NO_ERROR
366	ERROR_VIO_WAIT_FLAG
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
434	ERROR_VIO_LOCK
436	ERROR_VIO_INVALID_HANDLE

VioScrLock - Remarks

VioScrLock permits a process to determine if I/O to the physical screen buffer can take place. This prevents the process from writing to the physical buffer when the process is in the background. Processes must cooperate with the system in coordinating screen accesses.

Screen switching is disabled while the screen lock is in place. If a screen switch is suspended by a screen lock, and if the application holding the lock does not issue VioScrUnLock within a system-defined time limit, the screen switch occurs, and the process holding the lock is frozen in the background. A process should yield the screen lock as soon as possible, to avoid being frozen when running in the background. The timeout on the lock does not begin until a screen switch is requested.

When the screen lock is in effect and another thread in the same or different process (in the same session) issues VioScrLock, the second thread receives an error code. VioScrUnLock must be issued by the same thread that issued VioScrLock.

VioScrLock - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioScrollDown

VioScrollDown - Syntax

Scrolls the entire display buffer (or area specified within the display buffer) down.

```
#define INCL_VIO
#include <os2.h>

ULONG     TopRow;     /*  Top row to be scrolled. */
ULONG     LeftCol;    /*  Left column to be scrolled. */
ULONG     BotRow;     /*  Bottom row to be scrolled. */
ULONG     RightCol;   /*  Right column to be scrolled. */
ULONG     Lines;      /*  Number of lines. */
PBYTE     Cell;       /*  Cell to be written. */
HVIO      VioHandle;  /*  Vio presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioScrollDown(TopRow, LeftCol, BotRow,
       RightCol, Lines, Cell, VioHandle);
```

VioScrollDown Parameter - TopRow

TopRow (ULONG) - input
Top row to be scrolled.

VioScrollDown Parameter - LeftCol

LeftCol (ULONG) - input
Left column to be scrolled.

VioScrollDown Parameter - BotRow

BotRow (ULONG) - input
Bottom row to be scrolled.

VioScrollDown Parameter - RightCol

RightCol (ULONG) - input
Right column to be scrolled.

VioScrollDown Parameter - Lines

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the top of the screen area being scrolled. If 0 is specified, no lines are scrolled.

VioScrollDown Parameter - Cell

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioScrollDown Parameter - VioHandle

VioHandle (HVIO) - input
Vio presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioScrollDown Return Value - rc

rc (APIRET) - returns
Return code.

VioScrollDown returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollDown - Parameters

TopRow (ULONG) - input
Top row to be scrolled.

LeftCol (ULONG) - input
Left column to be scrolled.

BotRow (ULONG) - input
Bottom row to be scrolled.

RightCol (ULONG) - input
Right column to be scrolled.

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the top of the screen area being scrolled. If 0 is specified, no lines are scrolled.

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioHandle (HVIO) - input
Vio presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioScrollDown returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollDown - Remarks

TopRow = 0 and *LeftCol* = 0 identifies the top-left corner of the screen.

If a value greater than the maximum value is specified for *TopRow*, *LeftCol*, *BotRow*, *RightCol*, or *Lines*, the maximum value for that parameter is used.

If *TopRow* and *LeftCol* = 0, and if *BotRow*, *RightCol*, and *Lines* are greater than the screen lines, the entire screen is filled with the character-attribute pair defined by *Cell*.

VioScrollDown - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioScrollLeft

VioScrollLeft - Syntax

Scrolls the entire display buffer (or area specified within the display buffer) to the left.

```
#define INCL_VIO
#include <os2.h>

ULONG     TopRow;     /*  Top row to be scrolled. */
ULONG     LeftCol;    /*  Left column to be scrolled. */
ULONG     BotRow;     /*  Bottom row to be scrolled. */
ULONG     RightCol;   /*  Right column to be scrolled. */
ULONG     Lines;      /*  Number of lines. */
PBYTE     Cell;       /*  Cell to be written. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioScrollLeft(TopRow, LeftCol, BotRow,
       RightCol, Lines, Cell, VioHandle);
```

VioScrollLeft Parameter - TopRow

TopRow (ULONG) - input
Top row to be scrolled.

VioScrollLeft Parameter - LeftCol

LeftCol (ULONG) - input
Left column to be scrolled.

VioScrollLeft Parameter - BotRow

BotRow (ULONG) - input
Bottom row to be scrolled.

VioScrollLeft Parameter - RightCol

RightCol (ULONG) - input
Right column to be scrolled.

VioScrollLeft Parameter - Lines

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the right of the screen area being scrolled. If 0 is specified, no lines are scrolled.

VioScrollLeft Parameter - Cell

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioScrollLeft Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioScrollLeft Return Value - rc

rc (APIRET) - returns
Return code.

VioScrollLeft returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollLeft - Parameters

TopRow (ULONG) - input
Top row to be scrolled.

LeftCol (ULONG) - input
Left column to be scrolled.

BotRow (ULONG) - input
Bottom row to be scrolled.

RightCol (ULONG) - input
Right column to be scrolled.

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the right of the screen area being scrolled. If 0 is specified, no lines are scrolled.

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioScrollLeft returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollLeft - Remarks

TopRow = 0 and *LeftCol* = 0 identify the top-left corner of the screen.

If a value greater than the maximum value is specified for *TopRow*, *LeftCol*, *BotRow*, *RightCol*, or *Lines*, the maximum value for that parameter is used.

If *TopRow* and *LeftCol* = 0, and if *BotRow*, *RightCol*, and *Lines* are greater than the screen lines, the entire screen is filled with the character-attribute pair defined by *Cell*.

VioScrollLeft - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioScrollRight

VioScrollRight - Syntax

Scrolls the entire display buffer (or area specified within the display buffer) to the right.

```
#define INCL_VIO
#include <os2.h>

ULONG     TopRow;     /*  Top row to be scrolled. */
ULONG     LeftCol;    /*  Left column to be scrolled. */
ULONG     BotRow;     /*  Bottom row to be scrolled. */
ULONG     RightCol;   /*  Right column to be scrolled. */
ULONG     Lines;      /*  Number of lines. */
PBYTE     Cell;       /*  Cell to be written. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioScrollRight(TopRow, LeftCol, BotRow,
       RightCol, Lines, Cell, VioHandle);
```

VioScrollRight Parameter - TopRow

TopRow (ULONG) - input
Top row to be scrolled.

VioScrollRight Parameter - LeftCol

LeftCol (ULONG) - input
Left column to be scrolled.

VioScrollRight Parameter - BotRow

BotRow (ULONG) - input
Bottom row to be scrolled.

VioScrollRight Parameter - RightCol

RightCol (ULONG) - input
Right column to be scrolled.

VioScrollRight Parameter - Lines

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the left of the screen area being scrolled. If 0 is specified, no lines are scrolled.

VioScrollRight Parameter - Cell

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioScrollRight Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioScrollRight Return Value - rc

rc (APIRET) - returns
Return code.

VioScrollRight returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollRight - Parameters

TopRow (ULONG) - input
Top row to be scrolled.

LeftCol (ULONG) - input
Left column to be scrolled.

BotRow (ULONG) - input
Bottom row to be scrolled.

RightCol (ULONG) - input
Right column to be scrolled.

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the left of the screen area being scrolled. If 0 is specified, no lines are scrolled.

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioScrollRight returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollRight - Remarks

TopRow = 0 and *LeftCol* = 0 identify the top-left corner of the screen.

If a value greater than the maximum value is specified for *TopRow* , *LeftCol* , *BotRow* , *RightCol* , or *Lines* , the maximum value for that parameter is used.

If *TopRow* and *LeftCol* = 0, and if *BotRow*, *RightCol*, and *Lines* are greater than the screen lines, the entire screen is filled with the character-attribute pair defined by *Cell*.

VioScrollRight - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioScrollUp

VioScrollUp - Syntax

Scrolls up the entire display buffer (or area specified within the display buffer).

```
#define INCL_VIO
#include <os2.h>

ULONG     TopRow;     /*  Top row to be scrolled. */
ULONG     LeftCol;    /*  Left column to be scrolled. */
ULONG     BotRow;     /*  Bottom row to be scrolled. */
ULONG     RightCol;   /*  Right column to be scrolled. */
ULONG     Lines;      /*  Number of lines. */
PBYTE     Cell;       /*  Cell to be written. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioScrollUp(TopRow, LeftCol, BotRow,
       RightCol, Lines, Cell, VioHandle);
```

VioScrollUp Parameter - TopRow

TopRow (ULONG) - input
Top row to be scrolled.

VioScrollUp Parameter - LeftCol

LeftCol (ULONG) - input
Left column to be scrolled.

VioScrollUp Parameter - BotRow

BotRow (ULONG) - input
Bottom row to be scrolled.

VioScrollUp Parameter - RightCol

RightCol (ULONG) - input
Right column to be scrolled.

VioScrollUp Parameter - Lines

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the bottom of the screen area being scrolled. If 0 is specified, no lines are scrolled.

VioScrollUp Parameter - Cell

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioScrollUp Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioScrollUp Return Value - rc

rc (APIRET) - returns
Return code.

VioScrollUp returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollUp - Parameters

TopRow (ULONG) - input
Top row to be scrolled.

LeftCol (ULONG) - input
Left column to be scrolled.

BotRow (ULONG) - input
Bottom row to be scrolled.

RightCol (ULONG) - input
Right column to be scrolled.

Lines (ULONG) - input
Number of lines.

Number of lines to be inserted at the bottom of the screen area being scrolled. If 0 is specified, no lines are scrolled.

Cell (PBYTE) - input
Cell to be written.

Address of the character-attribute pair (2 or 4 bytes) used as a fill character on inserted lines.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioScrollUp returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioScrollUp - Remarks

TopRow = 0 and *LeftCol* = 0 identify the top-left corner of the screen.

If a value greater than the maximum value is specified for *TopRow*, *LeftCol*, *BotRow*, *RightCol*, or *Lines*, the maximum value for that parameter is used.

If *TopRow* and *LeftCol* = 0, and if *BotRow*, *RightCol*, and *Lines* are greater than the screen lines, the entire screen is filled with the character-attribute pair defined by *Cell*.

VioScrollUp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioScrUnLock

VioScrUnLock - Syntax

Releases ownership of (unlocks) the physical display buffer.

```
#define INCL_VIO
#include <os2.h>

HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioScrUnLock(VioHandle);
```

VioScrUnLock Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

Reserved. Must be 0.

VioScrUnLock Return Value - rc

rc (APIRET) - returns
Return code.

VioScrlUnLock returns one of the following values:

0	NO_ERROR
367	ERROR_VIO_UNLOCK
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioScrUnLock - Parameters

VioHandle (HVIO) - input
VIO presentation-space handle.

Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioScrlUnLock returns one of the following values:

0	NO_ERROR
367	ERROR_VIO_UNLOCK
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioScrUnLock - Remarks

This call releases the screen lock that is set by VioScrLock. The VioScrUnLock call must be issued by the same thread that issued VioScrLock.

VioScrUnLock - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetAnsi

VioSetAnsi - Syntax

Activates or deactivates ANSI support.

```
#define INCL_VIO
#include <os2.h>

ULONG     Indicator;  /*  On/Off indicator. */
```

```
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioSetAnsi(Indicator, VioHandle);
```

VioSetAnsi Parameter - Indicator

Indicator (ULONG) - input
On/Off indicator.

Equals 1 to activate ANSI support, or 0 to deactivate ANSI.

VioSetAnsi Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioSetAnsi Return Value - rc

rc (APIRET) - returns
Return code.

VioSetAnsi returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioSetAnsi - Parameters

Indicator (ULONG) - input
On/Off indicator.

Equals 1 to activate ANSI support, or 0 to deactivate ANSI.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioSetAnsi returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioSetAnsi - Remarks

For ANSI support, "ON" is the default.

VioSetAnsi - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetCp

VioSetCp - Syntax

Sets the code page used to display text data on the screen for the specified handle.

```
#define INCL_VIO
#include <os2.h>

ULONG     Reserved;     /*  Reserved.  Must be 0. */
USHORT    CodePageID;   /*  Code-page ID. */
HVIO      VioHandle;    /*  VIO presentation-space handle. */
APIRET    rc;           /*  Return code. */

rc = VioSetCp(Reserved, CodePageID, VioHandle);
```

VioSetCp Parameter - Reserved

Reserved (ULONG) - input
Reserved. Must be 0.

VioSetCp Parameter - CodePageID

CodePageID (USHORT) - input
Code-page ID.

The *CodePageID* must be a known code page.

VioSetCp Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioSetCp Return Value - rc

rc (APIRET) - returns
Return code.

VioSetCp returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
469	ERROR_VIO_BAD_CP

VioSetCp - Parameters

Reserved (ULONG) - input
Reserved. Must be 0.

CodePageID (USHORT) - input
Code-page ID.

The *CodePageID* must be a known code page.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioSetCp returns one of the following values:

0	NO_ERROR

355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
469	ERROR_VIO_BAD_CP

VioSetCp - Remarks

The specified code page applies to all new characters. How VioSetCp acts on characters already in the video buffer is undefined.

VioSetCp - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetCurPos

VioSetCurPos - Syntax

Sets the cursor's coordinates on the screen.

```
#define INCL_VIO
#include <os2.h>

ULONG     Row;
USHORT    Column;
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioSetCurPos(Row, Column, VioHandle);
```

VioSetCurPos Parameter - Row

Row (ULONG) - input
New cursor row position, where 0 is the top row.

VioSetCurPos Parameter - Column

Column (USHORT) - input

New cursor column position, where 0 is the leftmost column.

VioSetCurPos Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioSetCurPos Return Value - rc

rc (APIRET) - returns
Return code.

VioSetCurPos returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioSetCurPos - Parameters

Row (ULONG) - input
New cursor row position, where 0 is the top row.

Column (USHORT) - input

New cursor column position, where 0 is the leftmost column.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioSetCurPos returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE

358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioSetCurPos - Topics

Select an item:
Syntax
Parameters
Returns
Glossary

VioSetCurType

VioSetCurType - Syntax

Sets the cursor type.

```
#define INCL_VIO
#include <os2.h>

PVIOCURSORINFO    CursorData;  /*  Cursor characteristics. */
HVIO              VioHandle;   /*  VIO presentation-space handle. */
APIRET            rc;          /*  Return code. */

rc = VioSetCurType(CursorData, VioHandle);
```

VioSetCurType Parameter - CursorData

CursorData (PVIOCURSORINFO) - input
Cursor characteristics.

Address of the cursor-characteristics structure.

VioSetCurType Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioSetCurType Return Value - rc

rc (APIRET) - returns
Return code.

VioSetCurType returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
356	ERROR_VIO_WIDTH
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
465	ERROR_VIO_DETACHED

VioSetCurType - Parameters

CursorData (PVIOCURSORINFO) - input
Cursor characteristics.

Address of the cursor-characteristics structure.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioSetCurType returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
356	ERROR_VIO_WIDTH
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
465	ERROR_VIO_DETACHED

VioSetCurType - Remarks

To set the cursor start line (*yStart*) and the cursor end line (*cEnd*) independent of the number of scan lines for each character cell, you can specify these parameters as percentages. The OS/2 operating system then calculates the physical start and end scan lines, respectively, by multiplying the percentage specified for the parameter by the total number of scan lines in the character cell and rounding to the nearest scan line. Percentages are specified as negative values (or 0) in the range 0 through -100. Specifying *yStart* = -90 and *cEnd* = -100 requests a cursor that occupies the bottom 10% of the character cell.

The actual appearance of the cursor is hardware dependent. The video hardware might not support the specified parameters.

VioSetCurType - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetDeviceCellSize

VioSetDeviceCellSize - Syntax

Sets the size of the current character cell in pels.

```
#define INCL_VIO
#include <os2.h>

ULONG     Height;  /*  The height of the character cell in pels. */
ULONG     Width;   /*  The width of the character cell in pels. */
HVIO      hvps;    /*  VIO presentation-space handle. */
APIRET    rc;      /*  Return code. */

rc = VioSetDeviceCellSize(Height, Width, hvps);
```

VioSetDeviceCellSize Parameter - Height

Height (ULONG) - input
The height of the character cell in pels.

VioSetDeviceCellSize Parameter - Width

Width (ULONG) - input
The width of the character cell in pels.

VioSetDeviceCellSize Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session or a value returned by apiref refid='viocrps' form='textonly'..

VioSetDeviceCellSize Return Value - rc

rc (APIRET) - returns
Return code.

VioSetDeviceCellSize returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioSetDeviceCellSize - Parameters

Height (ULONG) - input
The height of the character cell in pels.

Width (ULONG) - input
The width of the character cell in pels.

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session or a value returned by apiref refid='viocrps' form='textonly'..

rc (APIRET) - returns
Return code.

VioSetDeviceCellSize returns one of the following values:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioSetDeviceCellSize - Remarks

If the device does not support the specified cell size, the cell size closest to the specified size is used. VioGetDeviceCellSize can be used to find the actual size selected. In OS/2 2.x, *hvps* cannot be 0.

VioSetDeviceCellSize - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetMode

VioSetMode - Syntax

Sets the mode of the display.

```
#define INCL_VIO
#include <os2.h>

PVIOMODEINFO    ModeData;   /*  Mode characteristics. */
HVIO            VioHandle;  /*  VIO presentation-space handle. */
APIRET          rc;         /*  Return code. */

rc = VioSetMode(ModeData, VioHandle);
```

VioSetMode Parameter - ModeData

ModeData (PVIOMODEINFO) - input
Mode characteristics.

Address of the mode characteristics structure.

VioSetMode Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioSetMode Return Value - rc

rc (APIRET) - returns
Return code.

VioSetMode returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

438	ERROR_VIO_INVALID_LENGTH
467	ERROR_VIO_FONT

VioSetMode - Parameters

ModeData (PVIOMODEINFO) - input
Mode characteristics.

Address of the mode characteristics structure.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioSetMode returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH
467	ERROR_VIO_FONT

VioSetMode - Remarks

VioSetMode initializes the cursor position and type.

VioSetMode does not clear the screen if the new and old modes are compatible. To clear the screen, use one of the VioScrollxx calls.

Assuming that no target display configuration for VioSetMode is selected, the mode is set on the primary configuration. If the primary configuration does not support the specified mode, the mode is set on the secondary configuration.

VioSetMode - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetOrigin

VioSetOrigin - Syntax

Sets the position at which the presentation space maps to the window.

```
#define INCL_VIO
#include <os2.h>

ULONG     Row;     /*  The top-most row shown in the window */
ULONG     Column;  /*  The left-most column shown in the window. */
HVIO      hvps;    /*  VIO presentation-space handle. */
APIRET    rc;      /*  Return code. */

rc = VioSetOrigin(Row, Column, hvps);
```

VioSetOrigin Parameter - Row

Row (ULONG) - input
The top-most row shown in the window

VioSetOrigin Parameter - Column

Column (ULONG) - input
The left-most column shown in the window.

VioSetOrigin Parameter - hvps

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session or a value returned by VioCreatePS.

VioSetOrigin Return Value - rc

rc (APIRET) - returns
Return code.

VioSetOrigin returns one of the following values:

0	NO_ERROR
358	ERROR_VIO_ROW
359	ERROR_VIO_COL

421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioSetOrigin - Parameters

Row (ULONG) - input
The top-most row shown in the window

Column (ULONG) - input
The left-most column shown in the window.

hvps (HVIO) - input
VIO presentation-space handle.

This is either 0 to indicate the default VIO session or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioSetOrigin returns one of the following values:

0	NO_ERROR
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioSetOrigin - Remarks

VioSetOrigin is used when the presentation space is larger than the window size to control which part of the presentation space is displayed. It does not, itself, cause any output to be displayed.

In OS/2 2.x, *hvps* cannot be 0.

VioSetOrigin - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioSetState

VioSetState - Syntax

Sets one or more of the following states:

- Blink and background intensity switch
- Color registers
- Overscan (border) color
- Palette registers
- Target VioSetMode display configuration
- Underline location

```
#define INCL_VIO
#include <os2.h>

PVIOD     RequestBlock;
HVIO      VioHandle;     /*  VIO presentation-space handle */
APIRET    rc;            /*  Return code. */

rc = VioSetState(RequestBlock, VioHandle);
```

VioSetState Parameter - RequestBlock

RequestBlock (PVIOD) - input
Address of the video state structures (consisting of six different structures, depending on the request type):

	Type	Definition
0		Set palette registers
1		Set overscan (border) color
2		Set blink/background intensity switch
3		Set color registers
4		Reserved
5		Set underline location
6		Set target VioSetMode display configuration

The six structures, depending on request type, are:

- VIOPALSTATE
- VIOOVERSCAN
- VIOINTENSITY
- VIOCOLORREG
- VIOSETULINELOC
- VIOSETTARGET

VioSetState Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle

Reserved. Must be 0.

VioSetState Return Value - rc

rc (APIRET) - returns
Return code.

VioSetState returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioSetState - Parameters

RequestBlock (PVIOD) - input
Address of the video state structures (consisting of six different structures, depending on the request type):

Type	Definition
0	Set palette registers
1	Set overscan (border) color
2	Set blink/background intensity switch
3	Set color registers
4	Reserved
5	Set underline location
6	Set target VioSetMode display configuration

The six structures, depending on request type, are:

- VIOPALSTATE
- VIOOVERSCAN
- VIOINTENSITY
- VIOCOLORREG
- VIOSETULINELOC
- VIOSETTARGET

VioHandle (HVIO) - input
VIO presentation-space handle

Reserved. Must be 0.

rc (APIRET) - returns
Return code.

VioSetState returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE
438	ERROR_VIO_INVALID_LENGTH

VioSetState - Remarks

VioSetState allows setting of hardware-dependent features. Not all video hardware will honor these settings.

VioSetState - Topics

Select an item:
Syntax

Parameters
Returns
Remarks
Glossary

VioShowBuf

VioShowBuf - Syntax

Updates the physical display buffer with the logical video buffer (LVB).

```
#define INCL_VIO
#include <os2.h>

ULONG     OffSet;     /*  Offset into the LVB. */
ULONG     Length;
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code */

rc = VioShowBuf(OffSet, Length, VioHandle);
```

VioShowBuf Parameter - OffSet

OffSet (ULONG) - input
Offset into the LVB.

Starting offset, within the LVB, where the update to the screen is to start.

VioShowBuf Parameter - Length

Length (ULONG) - input
Length of the area to be updated to the screen.

VioShowBuf Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioShowBuf Return Value - rc

rc (APIRET) - returns
Return code

Return code descriptions are:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioShowBuf - Parameters

OffSet (ULONG) - input
Offset into the LVB.

Starting offset, within the LVB, where the update to the screen is to start.

Length (ULONG) - input
Length of the area to be updated to the screen.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code

Return code descriptions are:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
430	ERROR_VIO_ILLEGAL_DURING_POPUP
436	ERROR_VIO_INVALID_HANDLE

VioShowBuf - Remarks

VioShowBuf is ignored unless it is issued by a process that is currently executing in the foreground or in a window.

VioShowBuf - Topics

Select an item:
Syntax
Parameters
Returns

Remarks
Glossary

VioShowPS

VioShowPS - Syntax

Updates the display of the VIO presentation space.

```
#define INCL_VIO
#include <os2.h>

ULONG     Depth;  /*  Depth. */
ULONG     Width;  /*  Width. */
ULONG     Cell;   /*  Offset to first updated cell. */
HVIO      hvps;   /*  VIO presentation handle. */
APIRET    rc;     /*  Return code */

rc = VioShowPS(Depth, Width, Cell, hvps);
```

VioShowPS Parameter - Depth

Depth (ULONG) - input
Depth.

The depth of the updated rectangle.

VioShowPS Parameter - Width

Width (ULONG) - input
Width.

The width of the updated rectangle.

VioShowPS Parameter - Cell

Cell (ULONG) - input
Offset to first updated cell.

The offset to the first updated cell. The offset of the top-left corner is zero.

VioShowPS Parameter - hvps

hvps (HVIO) - input
VIO presentation handle.

VIO presentation space handle. This is either zero to indicate the default VIO session or a value returned by VioCreatePS.

VioShowPS Return Value - rc

rc (APIRET) - returns
Return code

Return code descriptions are:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioShowPS - Parameters

Depth (ULONG) - input
Depth.

The depth of the updated rectangle.

Width (ULONG) - input
Width.

The width of the updated rectangle.

Cell (ULONG) - input
Offset to first updated cell.

The offset to the first updated cell. The offset of the top-left corner is zero.

hvps (HVIO) - input
VIO presentation handle.

VIO presentation space handle. This is either zero to indicate the default VIO session or a value returned by VioCreatePS.

rc (APIRET) - returns
Return code

Return code descriptions are:

0	NO_ERROR
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioShowPS - Remarks

This call is used to specify that part or all of the presentation space the logical buffer needs to be redrawn.

This call has the same function as VioShowBuf, but the area to update is specified differently.

VioShowPS - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioWrtCellStr

VioWrtCellStr - Syntax

Writes a string of character-attribute pairs (cells) to the display.

```
#define INCL_VIO
#include <os2.h>

ULONG     CellStr;    /*  String to be written. */
ULONG     Length;     /*  Length of string. */
ULONG     Row;        /*  Starting row position for output. */
ULONG     Column;     /*  Starting column position for output. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtCellStr(CellStr, Length, Row, Column,
       VioHandle);
```

VioWrtCellStr Parameter - CellStr

CellStr (ULONG) - input

String to be written.

Address of the string of character-attribute cells to be written.

VioWrtCellStr Parameter - Length

Length (ULONG) - input
Length of string.

Length, in bytes, of the string to be written. Each character-attribute cell is 2 or 4 bytes.

VioWrtCellStr Parameter - Row

Row (ULONG) - input
Starting row position for output.

VioWrtCellStr Parameter - Column

Column (ULONG) - input
Starting column position for output.

VioWrtCellStr Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtCellStr Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtCellStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS

436 ERROR_VIO_INVALID_HANDLE

VioWrtCellStr - Parameters

CellStr (ULONG) - input
String to be written.

Address of the string of character-attribute cells to be written.

Length (ULONG) - input
Length of string.

Length, in bytes, of the string to be written. Each character-attribute cell is 2 or 4 bytes.

Row (ULONG) - input
Starting row position for output.

Column (ULONG) - input
Starting column position for output.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtCellStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtCellStr - Remarks

If a string write gets to the end of the line and is not complete, it continues at the beginning of the next line. If the write gets to the end of the screen, it terminates.

VioWrtCellStr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioWrtCharStr

VioWrtCharStr - Syntax

Writes a character string to the display.

```
#define INCL_VIO
#include <os2.h>

PCH       CharStr;    /*  String to be written. */
ULONG     Length;     /*  Length of character string. */
ULONG     Row;        /*  Starting row position. */
ULONG     Column;     /*  Starting column position. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtCharStr(CharStr, Length, Row, Column,
       VioHandle);
```

VioWrtCharStr Parameter - CharStr

CharStr (PCH) - input
String to be written.

Address of the character string to be written.

VioWrtCharStr Parameter - Length

Length (ULONG) - input
Length of character string.

Length, in bytes, of the character string.

VioWrtCharStr Parameter - Row

Row (ULONG) - input
Starting row position.

VioWrtCharStr Parameter - Column

Column (ULONG) - input
Starting column position.

VioWrtCharStr Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtCharStr Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtCharStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtCharStr - Parameters

CharStr (PCH) - input
String to be written.

Address of the character string to be written.

Length (ULONG) - input
Length of character string.

Length, in bytes, of the character string.

Row (ULONG) - input
Starting row position.

Column (ULONG) - input
Starting column position.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtCharStr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW

359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtCharStr - Remarks

If a string write gets to the end of the line and is not complete, it continues at the beginning of the next line. If the write gets to the end of the screen, it terminates.

VioWrtCharStr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioWrtCharStrAtt

VioWrtCharStrAtt - Syntax

Writes a character string with repeated attributes to the display.

```
#define INCL_VIO
#include <os2.h>

PCH       CharStr;    /*  String to be written. */
ULONG     Length;     /*  Length, in bytes, of the character string. */
ULONG     Row;        /*  Starting row position. */
ULONG     Column;     /*  Starting column position. */
PBYTE     Attr;       /*  Attribute to be replicated. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtCharStrAtt(CharStr, Length, Row,
       Column, Attr, VioHandle);
```

VioWrtCharStrAtt Parameter - CharStr

CharStr (PCH) - input
String to be written.

Address of the character string to be written.

VioWrtCharStrAtt Parameter - Length

Length (ULONG) - input
Length, in bytes, of the character string.

VioWrtCharStrAtt Parameter - Row

Row (ULONG) - input
Starting row position.

VioWrtCharStrAtt Parameter - Column

Column (ULONG) - input
Starting column position.

VioWrtCharStrAtt Parameter - Attr

Attr (PBYTE) - input
Attribute to be replicated.

Address of the attributes (1 or 3 bytes) to be used in the display buffer for each character of the string written.

VioWrtCharStrAtt Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtCharStrAtt Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtCharStrAtt returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtCharStrAtt - Parameters

CharStr (PCH) - input
String to be written.

Address of the character string to be written.

Length (ULONG) - input
Length, in bytes, of the character string.

Row (ULONG) - input
Starting row position.

Column (ULONG) - input
Starting column position.

Attr (PBYTE) - input
Attribute to be replicated.

Address of the attributes (1 or 3 bytes) to be used in the display buffer for each character of the string written.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtCharStrAtt returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtCharStrAtt - Remarks

If a string write gets to the end of the line and is not complete, it continues at the beginning of the next line. If the write gets to the end of the screen, it terminates.

VioWrtCharStrAtt - Topics

Select an item:

Syntax
Parameters
Returns
Remarks
Glossary

VioWrtNAttr

VioWrtNAttr - Syntax

Writes an attribute to the display a specified number of times.

```
#define INCL_VIO
#include <os2.h>

PBYTE     Attr;       /*  Attribute to be written. */
ULONG     Times;      /*  Repeat count. */
ULONG     Row;        /*  Starting row position. */
ULONG     Column;     /*  Starting column position. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtNAttr(Attr, Times, Row, Column,
       VioHandle);
```

VioWrtNAttr Parameter - Attr

Attr (PBYTE) - input
Attribute to be written.

Address of the attributes (1 or 3 bytes) to be written.

VioWrtNAttr Parameter - Times

Times (ULONG) - input
Repeat count.

The number of times to write the attribute.

VioWrtNAttr Parameter - Row

Row (ULONG) - input
Starting row position.

VioWrtNAttr Parameter - Column

Column (ULONG) - input
Starting column position.

VioWrtNAttr Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtNAttr Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtNAttr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtNAttr - Parameters

Attr (PBYTE) - input
Attribute to be written.

Address of the attributes (1 or 3 bytes) to be written.

Times (ULONG) - input
Repeat count.

The number of times to write the attribute.

Row (ULONG) - input
Starting row position.

Column (ULONG) - input
Starting column position.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtNAttr returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtNAttr - Remarks

If a repeated write gets to the end of the line and is not complete, it continues at the beginning of the next line. If the write gets to the end of the screen, it terminates.

VioWrtNAttr - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioWrtNCell

VioWrtNCell - Syntax

Writes a cell (character-attribute pair) to the display a specified number of times.

```
#define INCL_VIO
#include <os2.h>

PBYTE     Cell;       /*  Attribute to be written. */
ULONG     Times;      /*  Repeat count. */
ULONG     Row;        /*  Starting row position. */
ULONG     Column;     /*  Starting column position. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtNCell(Cell, Times, Row, Column,
       VioHandle);
```

VioWrtNCell Parameter - Cell

Cell (PBYTE) - input
Attribute to be written.

The address of the character-attribute cell (2 or 4 bytes) to be written.

VioWrtNCell Parameter - Times

Times (ULONG) - input
Repeat count.

The number of times to write the attribute.

VioWrtNCell Parameter - Row

Row (ULONG) - input
Starting row position.

VioWrtNCell Parameter - Column

Column (ULONG) - input
Starting column position.

VioWrtNCell Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtNCell Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtNCell returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtNCell - Parameters

Cell (PBYTE) - input
Attribute to be written.

The address of the character-attribute cell (2 or 4 bytes) to be written.

Times (ULONG) - input
Repeat count.

The number of times to write the attribute.

Row (ULONG) - input
Starting row position.

Column (ULONG) - input
Starting column position.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtNCell returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtNCell - Remarks

If a repeated write gets to the end of the line and is not complete, it continues at the beginning of the next line. If the write gets to the end of the screen, it terminates.

VioWrtNCell - Topics

Select an item:
Syntax
Parameters

Returns
Remarks
Glossary

VioWrtNChar

VioWrtNChar - Syntax

Writes a character to the display a specified number of times.

```
#define INCL_VIO
#include <os2.h>

PCH       Char;       /*  Character to be written. */
ULONG     Times;      /*  Repeat count. */
ULONG     Row;        /*  Starting row position. */
ULONG     Column;     /*  Starting column position. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtNChar(Char, Times, Row, Column,
       VioHandle);
```

VioWrtNChar Parameter - Char

Char (PCH) - input
Character to be written.

The address of the character to be written.

VioWrtNChar Parameter - Times

Times (ULONG) - input
Repeat count.

The number of times to write the attribute.

VioWrtNChar Parameter - Row

Row (ULONG) - input

Starting row position.

VioWrtNChar Parameter - Column

Column (ULONG) - input
Starting column position.

VioWrtNChar Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtNChar Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtNChar returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtNChar - Parameters

Char (PCH) - input
Character to be written.

The address of the character to be written.

Times (ULONG) - input
Repeat count.

The number of times to write the attribute.

Row (ULONG) - input
Starting row position.

Column (ULONG) - input
Starting column position.

VioHandle (HVIO) - input

VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtNChar returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtNChar - Remarks

If a repeated write gets to the end of the line and is not complete, it continues at the beginning of the next line. If the write gets to the end of the screen, it terminates.

VioWrtNChar - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

VioWrtTTY

VioWrtTTY - Syntax

Writes a character string to the display, starting at the current cursor position. At the completion of the write, the cursor is moved to the first position beyond the end of the string.

```
#define INCL_VIO
#include <os2.h>

PCH       CharStr;    /*  String to be written. */
ULONG     Length;     /*  Length of string. */
HVIO      VioHandle;  /*  VIO presentation-space handle. */
APIRET    rc;         /*  Return code. */

rc = VioWrtTTY(CharStr, Length, VioHandle);
```

VioWrtTTY Parameter - CharStr

CharStr (PCH) - input
String to be written.

The address of the string to be written.

VioWrtTTY Parameter - Length

Length (ULONG) - input
Length of string.

The length of the character string in bytes.

VioWrtTTY Parameter - VioHandle

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

VioWrtTTY Return Value - rc

rc (APIRET) - returns
Return code.

VioWrtTTY returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtTTY - Parameters

CharStr (PCH) - input
String to be written.

The address of the string to be written.

Length (ULONG) - input
Length of string.

The length of the character string in bytes.

VioHandle (HVIO) - input
VIO presentation-space handle.

This must be 0, unless the caller is a Presentation Manager application; in this case, it must be the value returned by VioCreatePS.

rc (APIRET) - returns
Return code.

VioWrtTTY returns one of the following values:

0	NO_ERROR
355	ERROR_VIO_MODE
421	ERROR_VIO_INVALID_PARMS
436	ERROR_VIO_INVALID_HANDLE

VioWrtTTY - Remarks

If a string write gets to the end of the line and is not complete, the string write continues at the beginning of the next line. If the write gets to the end of the screen, the screen is scrolled, and the write continues until completed.

The character's carriage return, line feed, backspace, tab, and bell are treated as commands rather than printable characters. Backspace is a nondestructive backspace. Tabs are expanded to provide standard 8-byte-wide fields. VioWrtTTY is the only video function affected by ANSI.

Characters are written using the current attribute defined by ANSI or the default value of 7.

VioWrtTTY - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Glossary

Data Types

The following are all the data types used by the Control Program. They are listed in alphabetical order.

APIRET

Unsigned integer in the range 0 through 4 294 967 295.

```
typedef unsigned long APIRET;
```

AVAILDATA

Four-byte buffer in which the system returns the number of bytes that were available in the named pipe.

```
typedef struct _AVAILDATA {
  USHORT     cbpipe;     /*  The number of bytes that were buffered in the pipe, including the message-header bytes
mined. */
  USHORT     cbmessage;  /*  Number of bytes in the current message. */
} AVAILDATA;

typedef AVAILDATA *PAVAILDATA;
```

AVAILDATA Field - cbpipe

cbpipe (USHORT)
The number of bytes that were buffered in the pipe, including the message-header bytes and bytes that have been examined.

AVAILDATA Field - cbmessage

cbmessage (USHORT)
Number of bytes in the current message.

A value of 0 indicates a byte-stream pipe.

BANKINFO

Bank information data structure.

```
typedef struct _BANKINFO {
  ULONG      ulBankLength;     /*  Length of the bank. */
```

```
  USHORT     usBank;           /*  Current bank. */
  USHORT     usVideoModeType;  /*  Current video mode type. */
  USHORT     usReadWriteMode;  /*  Read/Write bank mode. */
} BANKINFO;
```

BANKINFO Field - ulBankLength

ulBankLength (ULONG)
Length of the bank.

Represents the combined length of all parameter packet fields. This is a required field for all calls, including those made by way of the 32-bit DosDevIOCtl, with a minimum packet length of 10 bytes.

BANKINFO Field - usBank

usBank (USHORT)
Current bank.

The current bank value to be set/returned. The bank size is 64KB regardless of the value specified for the video mode type field.

BANKINFO Field - usVideoModeType

usVideoModeType (USHORT)
Current video mode type.

Defines the adapter video mode type to the device driver, and has one of the following values:

0	Text Mode
1	Planar Mode
2	Linear Mode

This field is required.

BANKINFO Field - usReadWriteMode

usReadWriteMode (USHORT)
Read/Write bank mode.

Specifies what bank type is to be set/returned, and has one of the following values:

0	Read
1	Write

This field is required.

BIOSPARAMETERBLOCK

BIOS Parameter Block (BPB).

```
typedef struct _BIOSPARAMETERBLOCK {
  USHORT     usBytesPerSector;                /*  Number of bytes per sector. */
  BYTE       bSectorsPerCluster;              /*  Number of sectors per cluster. */
  USHORT     usReservedSectors;               /*  Number of reserved sectors. */
  BYTE       cFATs;                           /*  Number of FATs. */
  USHORT     cRootEntries;                    /*  Number of root directory entries. */
  USHORT     cSectors;                        /*  Number of sectors. */
  BYTE       bMedia;                          /*  Media descriptor. */
  USHORT     usSectorsPerFAT;                 /*  Number of secctors per FAT. */
  USHORT     usSectorsPerTrack;               /*  Number of sectors per track. */
  USHORT     cHeads;                          /*  Number of heads. */
  ULONG      cHiddenSectors;                  /*  Number of hidden sectors. */
  ULONG      cLargeSectors;                   /*  Number of large sectors. */
  BYTE       abReserved[6];  /*  Reserved. */
  USHORT     cCylinders;                      /*  Number of cylinders defined for the physical device. */
  BYTE       bDeviceType;                     /*  Physical layout of the specified device. */
  USHORT     fsDeviceAttr;                    /*  A bit field that returns flag information about the specified
} BIOSPARAMETERBLOCK;

typedef BIOSPARAMETERBLOCK *PBIOSPARAMETERBLOCK;
```

BIOSPARAMETERBLOCK Field - usBytesPerSector

usBytesPerSector (USHORT)
Number of bytes per sector.

BIOSPARAMETERBLOCK Field - bSectorsPerCluster

bSectorsPerCluster (BYTE)
Number of sectors per cluster.

BIOSPARAMETERBLOCK Field - usReservedSectors

usReservedSectors (USHORT)
Number of reserved sectors.

BIOSPARAMETERBLOCK Field - cFATs

cFATs (BYTE)
Number of FATs.

BIOSPARAMETERBLOCK Field - cRootEntries

cRootEntries (USHORT)
Number of root directory entries.

BIOSPARAMETERBLOCK Field - cSectors

cSectors (USHORT)
Number of sectors.

BIOSPARAMETERBLOCK Field - bMedia

bMedia (BYTE)
Media descriptor.

BIOSPARAMETERBLOCK Field - usSectorsPerFAT

usSectorsPerFAT (USHORT)
Number of secctors per FAT.

BIOSPARAMETERBLOCK Field - usSectorsPerTrack

usSectorsPerTrack (USHORT)
Number of sectors per track.

BIOSPARAMETERBLOCK Field - cHeads

cHeads (USHORT)
Number of heads.

BIOSPARAMETERBLOCK Field - cHiddenSectors

cHiddenSectors (ULONG)
Number of hidden sectors.

BIOSPARAMETERBLOCK Field - cLargeSectors

cLargeSectors (ULONG)
Number of large sectors.

BIOSPARAMETERBLOCK Field - abReserved[6]

abReserved[6] (BYTE)
Reserved.

BIOSPARAMETERBLOCK Field - cCylinders

cCylinders (USHORT)
Number of cylinders defined for the physical device.

BIOSPARAMETERBLOCK Field - bDeviceType

bDeviceType (BYTE)
Physical layout of the specified device.

0
: 48 TPI low-density diskette drive

1
: 96 TPI high-density diskette drive

2
: Small (3.5-inch) 720KB drive

3
: 8-inch single-density diskette drive

4
: 8-inch double-density diskette drive

5
: Fixed disk

6
: Tape drive

7
: Other (includes 1.44MB 3.5-inch diskette drive)

8

R/W optical disk

9

3.5-inch 4.0MB diskette drive (2.88MB formatted)

BIOSPARAMETERBLOCK Field - fsDeviceAttr

fsDeviceAttr (USHORT)

A bit field that returns flag information about the specified drive.

Bit 0

Removable Media flag.

0	Media is removable.
1	Media cannot be removed.

Bit 1

Changeline flag.

0	The physical device driver returns the value 0, *Unsure if media has changed*, from the Media Check function.
1	Device support determines that the media was removed since the last I/O operation.

BOOL

Boolean.

Valid values are:

- FALSE, which is 0
- TRUE, which is 1

```
typedef unsigned long BOOL;
```

BOOL32

Boolean.

Valid values are:

- FALSE, which is 0
- TRUE, which is 1

```
typedef unsigned long BOOL32;
```

BYTE

A byte.

```
typedef unsigned char BYTE;
```

CHAR

Single-byte character.

```
#define CHAR char
```

COLOR

Color value.

```
typedef LONG COLOR;
```

CONTEXTRECORD

This is the machine specific register contents for the thread at the time of the exception.

Note that only the register sets specified by *ContextFlags* contain valid data. Conversely, only registers specified in *ContextFlags* will be restored if an exception is handled.

```
typedef struct _CONTEXTRECORD {
  ULONG     ContextFlags;  /*  Flags which control the contents of the context record. */
  ULONG     ctx_env[7];
  FPREG     ctx_stack[8];
  ULONG     ctx_SegGs;     /*  GS register. */
  ULONG     ctx_SegFs;     /*  FS register. */
  ULONG     ctx_SegEs;     /*  ES register. */
  ULONG     ctx_SegDs;     /*  DS register. */
  ULONG     ctx_RegEdi;    /*  EDI register. */
  ULONG     ctx_RegEsi;    /*  ESI register. */
  ULONG     ctx_RegEax;    /*  EAX register. */
  ULONG     ctx_RegEbx;    /*  EBX register. */
  ULONG     ctx_RegEcx;    /*  ECX register. */
  ULONG     ctx_RegEdx;    /*  EDX register. */
  ULONG     ctx_RegEbp;    /*  EBP register. */
  ULONG     ctx_RegEip;    /*  EIP register. */
  ULONG     ctx_SegCs;     /*  CS register. */
  ULONG     ctx_EFlags;    /*  EFLAGS register. */
  ULONG     ctx_RegEsp;    /*  ESP register. */
  ULONG     ctx_SegSs;     /*  SS register. */
} CONTEXTRECORD;

typedef CONTEXTRECORD *PCONTEXTRECORD;
```

CONTEXTRECORD Field - ContextFlags

ContextFlags (ULONG)
Flags which control the contents of the context record.

Possible values are shown in the following list:

CONTEXT_CONTROL (0x00000001)
SS:ESP, CS:EIP, EFLAGS and EBP

CONTEXT_INTEGER (0x00000002)
EAX, EBX, ECX, EDX, ESI and EDI

CONTEXT_SEGMENTS (0x00000004)
DS, ES, FS, and GS

CONTEXT_FLOATING_POINT (0x00000008)
numeric coprocessor state

CONTEXT_FULL
CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_SEGMENTS | CONTEXT_FLOATING_POINT

If the context record is used as an input parameter, then for each portion of the context record controlled by a flag whose value is set, it is assumed that the portion of the context record contains valid context. If the context record is being used to modify a thread's context, then only that portion of the thread's context will be modified.

If the context record is used as an input/output parameter to capture the context of a thread, then only those portions of the thread's context corresponding to set flags will be returned.

CONTEXTRECORD Field - ctx_env[7]

ctx_env[7] (ULONG)
This parameter is used only when *ContextFlags* is set to CONTEXT_FLOATING_POINT.

CONTEXTRECORD Field - ctx_stack[8]

ctx_stack[8] (FPREG)
This parameter is used only when *ContextFlags* is set to CONTEXT_FLOATING_POINT.

CONTEXTRECORD Field - ctx_SegGs

ctx_SegGs (ULONG)
GS register.

This parameter is used only when *ContextFlags* is set to CONTEXT_SEGMENTS.

CONTEXTRECORD Field - ctx_SegFs

ctx_SegFs (ULONG)
FS register.

This parameter is used only when *ContextFlags* is set to CONTEXT_SEGMENTS.

CONTEXTRECORD Field - ctx_SegEs

ctx_SegEs (ULONG)
ES register.

This parameter is used only when *ContextFlags* is set to CONTEXT_SEGMENTS.

CONTEXTRECORD Field - ctx_SegDs

ctx_SegDs (ULONG)
DS register.

This parameter is used only when *ContextFlags* is set to CONTEXT_SEGMENTS.

CONTEXTRECORD Field - ctx_RegEdi

ctx_RegEdi (ULONG)
EDI register.

This parameter is used only when *ContextFlags* is set to CONTEXT_INTEGER.

CONTEXTRECORD Field - ctx_RegEsi

ctx_RegEsi (ULONG)
ESI register.

This parameter is used only when *ContextFlags* is set to CONTEXT_INTEGER.

CONTEXTRECORD Field - ctx_RegEax

ctx_RegEax (ULONG)
EAX register.

This parameter is used only when *ContextFlags* is set to CONTEXT_INTEGER.

CONTEXTRECORD Field - ctx_RegEbx

ctx_RegEbx (ULONG)
EBX register.

This parameter is used only when *ContextFlags* is set to CONTEXT_INTEGER.

CONTEXTRECORD Field - ctx_RegEcx

ctx_RegEcx (ULONG)
ECX register.

This parameter is used only when *ContextFlags* is set to CONTEXT_INTEGER.

CONTEXTRECORD Field - ctx_RegEdx

ctx_RegEdx (ULONG)
EDX register.

This parameter is used only when *ContextFlags* is set to CONTEXT_INTEGER.

CONTEXTRECORD Field - ctx_RegEbp

ctx_RegEbp (ULONG)
EBP register.

This parameter is used only when *ContextFlags* is set to CONTEXT_CONTROL.

CONTEXTRECORD Field - ctx_RegEip

ctx_RegEip (ULONG)
EIP register.

This parameter is used only when *ContextFlags* is set to CONTEXT_CONTROL.

CONTEXTRECORD Field - ctx_SegCs

ctx_SegCs (ULONG)
CS register.

This parameter is used only when *ContextFlags* is set to CONTEXT_CONTROL.

CONTEXTRECORD Field - ctx_EFlags

ctx_EFlags (ULONG)
EFLAGS register.

This parameter is used only when *ContextFlags* is set to CONTEXT_CONTROL.

CONTEXTRECORD Field - ctx_RegEsp

ctx_RegEsp (ULONG)
ESP register.

This parameter is used only when *ContextFlags* is set to CONTEXT_CONTROL.

CONTEXTRECORD Field - ctx_SegSs

ctx_SegSs (ULONG)
SS register.

This parameter is used only when *ContextFlags* is set to CONTEXT_CONTROL.

CPUUTIL

Performance data returned by DosPerfSysCall.

```
typedef struct _CPUUTIL {
  ULONG     ulTimeLow;   /*  Low 32 bits of time stamp */
  ULONG     ulTimeHigh;  /*  High 32 bits of time stamp */
  ULONG     ulIdleLow;   /*  Low 32 bits of idle time */
  ULONG     ulIdleHigh;  /*  High 32 bits of idle time */
  ULONG     ulBusyLow;   /*  Low 32 bits of busy time */
  ULONG     ulBusyHigh;  /*  High 32 bits of busy time */
  ULONG     ulIntrLow;   /*  Low 32 bits of interrupt time */
  ULONG     ulIntrHigh;  /*  High 32 bits of interrupt time */
} CPUUTIL;

typedef CPUUTIL *PCPUUTIL;
```

CPUUTIL Field - ulTimeLow

ulTimeLow (ULONG)
Low 32 bits of time stamp

CPUUTIL Field - ulTimeHigh

ulTimeHigh (ULONG)
High 32 bits of time stamp

CPUUTIL Field - ulIdleLow

ulIdleLow (ULONG)
Low 32 bits of idle time

CPUUTIL Field - ulIdleHigh

ulIdleHigh (ULONG)
High 32 bits of idle time

CPUUTIL Field - ulBusyLow

ulBusyLow (ULONG)
Low 32 bits of busy time

CPUUTIL Field - ulBusyHigh

ulBusyHigh (ULONG)
High 32 bits of busy time

CPUUTIL Field - ulIntrLow

ulIntrLow (ULONG)
Low 32 bits of interrupt time

CPUUTIL Field - ulIntrHigh

ulIntrHigh (ULONG)
High 32 bits of interrupt time

COUNTRYCODE

Country code and code page.

```
typedef struct _COUNTRYCODE {
  ULONG     country;   /*  The binary value of the selected country code. */
  ULONG     codepage;  /*  The binary value of the selected code page identifier. */
} COUNTRYCODE;

typedef COUNTRYCODE *PCOUNTRYCODE;
```

The following table shows the country, country code, primary code page, and secondary code page identifier values:

Country	Country Code	Primary	Secondary
Asian English	099	437	850
Australia	061	437	850
Belgium	032	437	850
Bulgaria	359	915	850
Canadian French	002	863	850
Catalan	034	850	437
Czechoslovakia	042	852	850
Denmark	045	865	850
Finland	358	437	850
France	033	437	850
Germany	049	437	850
Hungary	036	852	850
Iceland	354	850	861
Italy	039	437	850
Japan	081	932	437, 850
Japan SAA	081	942	437, 850
Korea	082	934	437, 850

Korea SAA	082	944	437, 850
Latin America	003	437	850
Lithuania	370	921	850
Netherlands	031	437	850
Norway	047	865	850
People's Republic of China	086	1381, 1386	946
Poland	048	852	850
Portugal	351	860	850
Slovenia	386	852	850
Spain	034	437	850
Sweden	046	437	850
Switzerland	041	437	850
Taiwan	088	938	437, 850
Taiwan SAA	088	948	437, 850
Turkey	090	857	850
United Kingdom	044	437	850
United States	001	437	850
Yugoslavia	038	852	850

Note: Code pages 932, 934, 936, 938, 942, 944, 946, 948, 1381, and 1386 are supported only with the Asian version of the operating system on Asian hardware.

COUNTRYCODE Field - country

country (ULONG)
The binary value of the selected country code.

If *country* is set to 0, the country information of the default system country code is used.

COUNTRYCODE Field - codepage

codepage (ULONG)
The binary value of the selected code page identifier.

If *codepage* is set to 0, the country information for the current process code page of the caller is used.

COUNTRYINFO

Country information.

```
typedef struct _COUNTRYINFO {
  ULONG      country;                /*  Country code. */
  ULONG      codepage;               /*  Code page. */
  ULONG      fsDateFmt;              /*  Date format. */
  CHAR       szCurrency[5];          /*  Currency indicator, null terminated. */
  CHAR       szThousandsSeparator[2]; /*  Thousands separator, null terminated. */
  CHAR       szDecimal[2];           /*  Decimal separator, null terminated. */
  CHAR       szDateSeparator[2];     /*  Date separator, null terminated. */
  CHAR       szTimeSeparator[2];     /*  Time separator, null terminated. */
  UCHAR      fsCurrencyFmt;          /*  Bit fields for currency format. */
  UCHAR      cDecimalPlace;          /*  Binary number of decimal places used in the currency indication. */
  UCHAR      fsTimeFmt;              /*  Time format for file directory presentation. */
  USHORT     abReserved[2];          /*  Reserved; set to 0. */
  CHAR       szDataSeparator[2];     /*  Data list separator, null terminated. */
  USHORT     abReserved2[5];         /*  Reserved; set to 0. */
} COUNTRYINFO;

typedef COUNTRYINFO *PCOUNTRYINFO;
```

COUNTRYINFO Field - country

country (ULONG)
Country code.

COUNTRYINFO Field - codepage

codepage (ULONG)
Code page.

COUNTRYINFO Field - fsDateFmt

fsDateFmt (ULONG)
Date format.

Possible values are shown in the following list:

0	mm/dd/yy
1	dd/mm/yy
2	yy/mm/dd

COUNTRYINFO Field - szCurrency[5]

szCurrency[5] (CHAR)
Currency indicator, null terminated.

COUNTRYINFO Field - szThousandsSeparator[2]

szThousandsSeparator[2] (CHAR)
Thousands separator, null terminated.

COUNTRYINFO Field - szDecimal[2]

szDecimal[2] (CHAR)
Decimal separator, null terminated.

COUNTRYINFO Field - szDateSeparator[2]

szDateSeparator[2] (CHAR)
Date separator, null terminated.

COUNTRYINFO Field - szTimeSeparator[2]

szTimeSeparator[2] (CHAR)
Time separator, null terminated.

COUNTRYINFO Field - fsCurrencyFmt

fsCurrencyFmt (UCHAR)
Bit fields for currency format.

This field contains the following bit fields:

Bit	Description
0	Placement of the currency indicator. 0 currency indicator precedes money value 1 currency indicator follows money value
1	Number of spaces (0 or 1) between currency indicator and money value.
2	When this bit is set, ignore the first two bits; the currency indicator replaces the decimal indicator.

COUNTRYINFO Field - cDecimalPlace

cDecimalPlace (UCHAR)
Binary number of decimal places used in the currency indication.

COUNTRYINFO Field - fsTimeFmt

fsTimeFmt (UCHAR)
Time format for file directory presentation.

The following values are possible:

0	12 hour with "am" or "pm"
1	24 hour

COUNTRYINFO Field - abReserved[2]

abReserved[2] (USHORT)
Reserved; set to 0.

COUNTRYINFO Field - szDataSeparator[2]

szDataSeparator[2] (CHAR)
Data list separator, null terminated.

COUNTRYINFO Field - abReserved2[5]

abReserved2[5] (USHORT)
Reserved; set to 0.

CPID

Code page identification data structure.

```
typedef struct _CPID {
  USHORT     idCodePage;  /*  Current code page number. */
  USHORT     reserved;    /*  Reserved. */
```

```
} CPID;

typedef CPID *PCPID;
```

CPID Field - idCodePage

idCodePage (USHORT)
Current code page number.

CPID Field - reserved

reserved (USHORT)
Reserved.

Set to 0.

cvkcmd_s

Data associated with the kernel debugger communications protocol.

```
typedef struct _cvkcmd_s {
  USHORT          Cmd;       /*  Command */
  ULONG           Value;     /*  Value, command dependent. */
  ULONG           OffV;      /*  Command dependent, usually a linear address. */
  USHORT          SegV;      /*  Command dependent.  Usually slot number of thread. */
  USHORT          MTE;       /*  Module Table Entry */
  USHORT          PID;       /*  Process Identifier */
  USHORT          TID;       /*  Thread Identifier */
  USHORT          DBit;      /*  Flags from the CS selector. */
  RegSA_struc     Reg;       /*  Register save area */
  UCHAR           MemCache;  /*  Data area */
} cvkcmd_s;

typedef cvkcmd_s *cvkcmd_s;
```

cvkcmd_s Field - Cmd

Cmd (USHORT)
Command

One of the following:

Command	Code	Description	CVK_CMDSIZE_	CVK_RETSIZE_
CVK_CMD_RMEM	1	Read memory	18	20

CVK_CMD_RREG	3	Read registers	18	24 + sizeof(RegSa_ struc)
CVK_CMD_WMEM	4	Write memory	20	6
CVK_CMD_WREG	6	Write registers	20 + sizeof(RegSa_ struc)	2
CVK_CMD_RUN	7	Resume execution	6	0
CVK_CMD_KILL	8	Reboot victim machine	2	0
CVK_CMD_STEP	9	Single step	2	0
CVK_CMD_NUMTOBASE	13	Get object/segment information	14	14
CVK_CMD_LIBNAME	16	Get module information	6	6
CVK_CMD_RAW	20	Perform kernel debugger command	6	
CVK_CMD_DBIT	22	Get selector information	20	
CVK_CMD_RSTEP	23	Range step	10	0
CVK_CMD_SCANMTE	24	Scan module table	2	6
CVK_CMD_SCANTCB	25	Scan thread control blocks	6	10
CVK_CMD_SEL2LIN	26	Convert selector:offset to linear address.	18	6
CVK_CMD_LIN2SEL	27	Convert linear address to selector:offset.	18	12
CVK_CMD_OBJCOUNT	28	Get number of objects/segments in module	6	6
CVK_CMD_SCANOBJ	29	Scan object/segment table	14	10
CVK_CMD_SELINFO	30	Get selector information	18	20
CVK_CMD_RNPX	31	Read NPX state	18	128
CVK_CMD_WNPX	32	Write NPX state	128	60
CVK_CMD_ENA	33	Enable optional features	6	2
CVK_CMD_DIS	34	Disable optional features	6	2
CVK_CMD_PIREG	35	Register for PAGEIN notification	14	2
CVK_CMD_PIDRG	36	Deregister for PAGEIN notification	14	2

cvkcmd_s Field - Value

Value (ULONG)
Value, command dependent.

cvkcmd_s Field - OffV

OffV (ULONG)
Command dependent, usually a linear address.

cvkcmd_s Field - SegV

SegV (USHORT)
Command dependent. Usually slot number of thread.

cvkcmd_s Field - MTE

MTE (USHORT)
Module Table Entry

cvkcmd_s Field - PID

PID (USHORT)
Process Identifier

cvkcmd_s Field - TID

TID (USHORT)
Thread Identifier

cvkcmd_s Field - DBit

DBit (USHORT)

Flags from the CS selector.

cvkcmd_s Field - Reg

Reg (RegSA_struc)
Register save area

cvkcmd_s Field - MemCache

MemCache (UCHAR)
Data area

Data area used for variable length data in a command or response. Maximum size of the data is CVK_MEMCACHE_SIZE (512).

DATETIME

Date and time data structure.

```
typedef struct _DATETIME {
  UCHAR      hours;       /*  Current hour, using values 0 through 23. */
  UCHAR      minutes;     /*  Current minute, using values 0 through 59. */
  UCHAR      seconds;     /*  Current second, using values 0 through 59. */
  UCHAR      hundredths;  /*  Current hundredths of a second, using values 0 through 99. */
  UCHAR      day;         /*  Current day of the month, using values 1 through 31. */
  UCHAR      month;       /*  Current month of the year, using values 1 through 12. */
  USHORT     year;        /*  Current year. */
  SHORT      timezone;    /*  The difference in minutes between the current time zone and Greenwich Mean Time (G
  UCHAR      weekday;     /*  Current day of the week, using values 0 through 6. */
} DATETIME;

typedef DATETIME *PDATETIME;
```

DATETIME Field - hours

hours (UCHAR)
Current hour, using values 0 through 23.

DATETIME Field - minutes

minutes (UCHAR)
Current minute, using values 0 through 59.

DATETIME Field - seconds

seconds (UCHAR)
Current second, using values 0 through 59.

DATETIME Field - hundredths

hundredths (UCHAR)
Current hundredths of a second, using values 0 through 99.

DATETIME Field - day

day (UCHAR)
Current day of the month, using values 1 through 31.

DATETIME Field - month

month (UCHAR)
Current month of the year, using values 1 through 12.

DATETIME Field - year

year (USHORT)
Current year.

DATETIME Field - timezone

timezone (SHORT)
The difference in minutes between the current time zone and Greenwich Mean Time (GMT).

This value is positive for time zones west of Greenwich, England, and negative for time zones east of Greenwich. A value of -1 indicates that the time zone is undefined.

DATETIME Field - weekday

weekday (UCHAR)
Current day of the week, using values 0 through 6.

Sunday is equal to 0.

DCBINFO

Device control block information data structure.

```
typedef struct _DCBINFO {
  USHORT     usWriteTimeout;         /*  Time period used for Write Timeout processing. */
  USHORT     usReadTimeout;          /*  Time period used for Read Timeout processing. */
  BYTE       fbCtlHndShake;          /*  HandShake Control flag. */
  BYTE       fbFlowReplace;          /*  Flow Control flag. */
  BYTE       fbTimeout;              /*  Timeout flag. */
  BYTE       bErrorReplacementChar;  /*  Error Replacement Character. */
  BYTE       bBreakReplacementChar;  /*  Break Replacement Character. */
  BYTE       bXONChar;               /*  Character XON. */
  BYTE       bXOFFChar;              /*  Character XOFF. */
} DCBINFO;

typedef DCBINFO *PDCBINFO;
```

DCBINFO Field - usWriteTimeout

usWriteTimeout (USHORT)
Time period used for Write Timeout processing.

Specifies the time period used for Write Timeout processing. See Note 8 of ASYNC_SETDCBINFO. The value is in .01 second units based on zero (where 0 = .01 seconds).

DCBINFO Field - usReadTimeout

usReadTimeout (USHORT)
Time period used for Read Timeout processing.

Specifies the time period used for Read Timeout processing. See Note 9 of ASYNC_SETDCBINFO. The value is in .01 second units based on zero (where 0 =.01 seconds).

DCBINFO Field - fbCtlHndShake

fbCtlHndShake (BYTE)
HandShake Control flag.

Has the following bits:

Bits 0-1 — DTR Control mode. Has the following:

Bit 1	Bit 0	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Invalid input. Results in a general failure error.

Bit 2 — Reserved. Must be 0.

Bit 3 — Enable output handshaking using CTS

Bit 4 — Enable output handshaking using DSR

Bit 5 — Enable output handshaking using DCD

Bit 6 — Enable input sensitivity using DSR

Bit 7 — Reserved. Must be 0.

DCBINFO Field - fbFlowReplace

fbFlowReplace (BYTE)
Flow Control flag.

Has the following bits:

Bit 0 — Enable Automatic Transmit Flow Control (XON/XOFF)

Bit 1 — Enable Automatic Receive Flow Control (XON/XOFF)

Bit 2 — Enable error replacement character

Bit 3 — Enable null stripping (remove null bytes)

Bit 4 — Enable break replacement character

Bit 5 — Automatic Receive Flow Control:

0 = Normal
1 = Full-Duplex

Bits 6-7 — RTS Control mode. Has the following:

Bit 7	Bit 6	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Toggling on transmit

DCBINFO Field - fbTimeout

fbTimeout (BYTE)
Timeout flag.

Has the following bits:

Bit 0 — Enable Write Infinite Timeout processing

Bits 1-2 — Read Timeout processing. Has the following:

Bit 2	Bit 1	Description
0	1	Normal Read Timeout processing
1	0	Wait-For-Something, Read Timeout processing
1	1	No-Wait, Read Timeout processing

Bits 3-4 — Extended Hardware Buffering. Has the following:

Bit 4	Bit 3	Description
0	0	Not supported
0	1	Extended Hardware Buffering Disabled
1	0	Extended Hardware Buffering Enabled
1	1	Automatic Protocol Override

Bits 5-6 — Receive Trigger Level. Has the following:

Bit 6	Bit 5	Description
0	0	1 character
0	1	4 characters
1	0	8 characters
1	1	14 characters

Bit 7 — Transmit Buffer Load Count

0 = 1 character
1 = 16 characters

See ASYNC_SETDCBINFO for field definitions.

DCBINFO Field - bErrorReplacementChar

bErrorReplacementChar (BYTE)
Error Replacement Character.

Value in the range 00h-FFh. See note 5 of ASYNC_SETDCBINFO

DCBINFO Field - bBreakReplacementChar

bBreakReplacementChar (BYTE)
Break Replacement Character.

Value in the range 00h-FFh. See note 7 of ASYNC_SETDCBINFO

DCBINFO Field - bXONChar

bXONChar (BYTE)
Character XON.

Value in the range 00h-FFh. See note 2 of ASYNC_SETDCBINFO

DCBINFO Field - bXOFFChar

bXOFFChar (BYTE)
Character XOFF.

Value in the range 00h-FFh. See note 2 of ASYNC_SETDCBINFO

DENA2

DENA2 data structure.

```
typedef FEA2 DENA2;
```

DEVICEPARAMETERBLOCK

Device Parameter Block data structure.

```
typedef struct _DEVICEPARAMETERBLOCK {
  USHORT     reserved1;         /*  Reserved. */
  USHORT     cCylinders;        /*  Number of Cylinders on the physical drive. */
  USHORT     cHeads;            /*  Number of Heads on the physical drive. */
  USHORT     cSectorsPerTrack;  /*  Number of Sectors per track on the physical drive. */
  USHORT     reserved2;         /*  Reserved. */
  USHORT     reserved3;         /*  Reserved. */
  USHORT     reserved4;         /*  Reserved. */
  USHORT     reserved5;         /*  Reserved. */
} DEVICEPARAMETERBLOCK;

typedef DEVICEPARAMETERBLOCK *PDEVICEPARAMETERBLOCK;
```

DEVICEPARAMETERBLOCK Field - reserved1

reserved1 (USHORT)
Reserved.

DEVICEPARAMETERBLOCK Field - cCylinders

cCylinders (USHORT)
Number of Cylinders on the physical drive.

DEVICEPARAMETERBLOCK Field - cHeads

cHeads (USHORT)
Number of Heads on the physical drive.

DEVICEPARAMETERBLOCK Field - cSectorsPerTrack

cSectorsPerTrack (USHORT)
Number of Sectors per track on the physical drive.

DEVICEPARAMETERBLOCK Field - reserved2

reserved2 (USHORT)
Reserved.

DEVICEPARAMETERBLOCK Field - reserved3

reserved3 (USHORT)
Reserved.

DEVICEPARAMETERBLOCK Field - reserved4

reserved4 (USHORT)
Reserved.

DEVICEPARAMETERBLOCK Field - reserved5

reserved5 (USHORT)
Reserved.

DosDebug Buffer

DosDebug buffer structure.

```
typedef struct _DosDebug Buffer {
  ULONG      Pid;     /*  Debuggee Process ID */
  ULONG      Tid;     /*  Debuggee Thread ID */
  LONG       Cmd;     /*  Command or Notification */
  LONG       Value;   /*  Generic Data Value */
  ULONG      Addr;    /*  Debuggee Address */
  ULONG      Buffer;  /*  Debugger Buffer Address */
  ULONG      Len;     /*  Length of Range */
  ULONG      Index;   /*  Generic Identifier Index */
  ULONG      MTE;     /*  Module Table Entry Handle */
  ULONG      EAX;     /*  Register Set */
  ULONG      ECX;
  ULONG      EDX;
  ULONG      EBX;
  ULONG      ESP;
  ULONG      EBP;
  ULONG      ESI;
  ULONG      EDI;
  ULONG      EFlags;
  ULONG      EIP;
  ULONG      CSLim;   /*  Byte Granular Limits */
  ULONG      CSBase;  /*  Byte Granular Base */
  UCHAR      CSAcc;   /*  Access Bytes */
  UCHAR      CSAtr;   /*  Attribute Bytes */
  USHORT     CS;
  ULONG      DSLim;
  ULONG      DSBase;
  UCHAR      DSAcc;
  UCHAR      DSAtr;
  USHORT     DS;
  ULONG      ESLim;
  ULONG      ESBase;
  UCHAR      ESAcc;
  UCHAR      ESAtr;
  USHORT     ES;
  ULONG      FSLim;
  ULONG      FSBase;
  UCHAR      FSAcc;
  UCHAR      FSAtr;
  USHORT     FS;
  ULONG      GSLim;
  ULONG      GSBase;
  UCHAR      GSAcc;
  UCHAR      GSAtr;
  USHORT     GS;
  ULONG      SSLim;
  ULONG      SSBase;
  UCHAR      SSAcc;
  UCHAR      SSAtr;
  USHORT     SS;
} DosDebug Buffer;

typedef DosDebug Buffer *DBUGBUF;
```

DosDebug Buffer Field - Pid

Pid (ULONG)
Debuggee Process ID

DosDebug Buffer Field - Tid

Tid (ULONG)
Debuggee Thread ID

DosDebug Buffer Field - Cmd

Cmd (LONG)
Command or Notification

DosDebug Buffer Field - Value

Value (LONG)
Generic Data Value

DosDebug Buffer Field - Addr

Addr (ULONG)
Debuggee Address

DosDebug Buffer Field - Buffer

Buffer (ULONG)
Debugger Buffer Address

DosDebug Buffer Field - Len

Len (ULONG)
Length of Range

DosDebug Buffer Field - Index

Index (ULONG)
Generic Identifier Index

DosDebug Buffer Field - MTE

MTE (ULONG)
Module Table Entry Handle

DosDebug Buffer Field - EAX

EAX (ULONG)
Register Set

DosDebug Buffer Field - ECX

ECX (ULONG)

DosDebug Buffer Field - EDX

EDX (ULONG)

DosDebug Buffer Field - EBX

EBX (ULONG)

DosDebug Buffer Field - ESP

ESP (ULONG)

DosDebug Buffer Field - EBP

EBP (ULONG)

DosDebug Buffer Field - ESI

ESI (ULONG)

DosDebug Buffer Field - EDI

EDI (ULONG)

DosDebug Buffer Field - EFlags

EFlags (ULONG)

DosDebug Buffer Field - EIP

EIP (ULONG)

DosDebug Buffer Field - CSLim

CSLim (ULONG)
Byte Granular Limits

DosDebug Buffer Field - CSBase

CSBase (ULONG)
Byte Granular Base

DosDebug Buffer Field - CSAcc

CSAcc (UCHAR)
Access Bytes

DosDebug Buffer Field - CSAtr

CSAtr (UCHAR)
Attribute Bytes

DosDebug Buffer Field - CS

CS (USHORT)

DosDebug Buffer Field - DSLim

DSLim (ULONG)

DosDebug Buffer Field - DSBase

DSBase (ULONG)

DosDebug Buffer Field - DSAcc

DSAcc (UCHAR)

DosDebug Buffer Field - DSAtr

DSAtr (UCHAR)

DosDebug Buffer Field - DS

DS (USHORT)

DosDebug Buffer Field - ESLim

ESLim (ULONG)

DosDebug Buffer Field - ESBase

ESBase (ULONG)

DosDebug Buffer Field - ESAcc

ESAcc (UCHAR)

DosDebug Buffer Field - ESAtr

ESAtr (UCHAR)

DosDebug Buffer Field - ES

ES (USHORT)

DosDebug Buffer Field - FSLim

FSLim (ULONG)

DosDebug Buffer Field - FSBase

FSBase (ULONG)

DosDebug Buffer Field - FSAcc

FSAcc (UCHAR)

DosDebug Buffer Field - FSAtr

FSAtr (UCHAR)

DosDebug Buffer Field - FS

FS (USHORT)

DosDebug Buffer Field - GSLim

GSLim (ULONG)

DosDebug Buffer Field - GSBase

GSBase (ULONG)

DosDebug Buffer Field - GSAcc

GSAcc (UCHAR)

DosDebug Buffer Field - GSAtr

GSAtr (UCHAR)

DosDebug Buffer Field - GS

GS (USHORT)

DosDebug Buffer Field - SSLim

SSLim (ULONG)

DosDebug Buffer Field - SSBase

SSBase (ULONG)

DosDebug Buffer Field - SSAcc

SSAcc (UCHAR)

DosDebug Buffer Field - SSAtr

SSAtr (UCHAR)

DosDebug Buffer Field - SS

SS (USHORT)

EAOP2

EAOP2 data structure.

```
typedef struct _EAOP2 {
  PGEA2LIST     fpGEA2List;  /*  GEA set. */
  PFEA2LIST     fpFEA2List;  /*  FEA set. */
  ULONG         oError;      /*  Offset of FEA error. */
} EAOP2;

typedef EAOP2 *PEAOP2;
```

EAOP2 Field - fpGEA2List

fpGEA2List (PGEA2LIST)
GEA set.

EAOP2 Field - fpFEA2List

fpFEA2List (PFEA2LIST)
FEA set.

EAOP2 Field - oError

oError (ULONG)
Offset of FEA error.

EASIZEBUF

Maximum size of extended attributes (EAs).

```
typedef struct _EASIZEBUF {
  USHORT     cbMaxEASize;      /*  Maximum size of an EA. */
  ULONG      cbMaxEAListSize;  /*  Maximum size of the full EA list. */
} EASIZEBUF;

typedef EASIZEBUF *PEASIZEBUF;
```

EASIZEBUF Field - cbMaxEASize

cbMaxEASize (USHORT)
Maximum size of an EA.

EASIZEBUF Field - cbMaxEAListSize

cbMaxEAListSize (ULONG)
Maximum size of the full EA list.

ERRORID

Error identity.

```
typedef ULONG ERRORID;
```

EXCEPTIONREGISTRATIONRECORD

These structures are linked together to form a chain of exception handlers that are dispatched upon receipt of an exception. Exception handlers should not be registered directly from a high level language such as "C". This is the responsibility of the language runtime routine.

```
typedef struct _EXCEPTIONREGISTRATIONRECORD {
  STRUCT _EXCEPTIONREGISTRATIONRECORD      *prev_structure;    /*  Nested exception registration record structur
  _ERR                                     *ExceptionHandler;  /*  Pointer to the ERR function. */
} EXCEPTIONREGISTRATIONRECORD;

typedef EXCEPTIONREGISTRATIONRECORD *PEXCEPTIONREGISTRATIONRECORD;
```

EXCEPTIONREGISTRATIONRECORD Field - prev_structure

prev_structure (STRUCT _EXCEPTIONREGISTRATIONRECORD *)
Nested exception registration record structure.

This field should be treated as a C-language volatile field. That is, even though this field may be changed in ways unknown to your program, the intent of the original expression will be maintained.

EXCEPTIONREGISTRATIONRECORD Field - ExceptionHandler

ExceptionHandler (_ERR *)
Pointer to the ERR function.

This field must be treated as a C-language volatile field. That is, even though this field may be changed in ways unknown to your program, the intent of the original expression will be maintained.

The ERR function is defined below:

```
typedef ULONG APIENTRY _ERR (PEXECPTIONREPORTRECORD,
                             struct _EXCEPTIONREGISTRATIONRECORD *,
                             PCONTEXTRECORD,
                             PVOID);

typedef _ERR *ERR;
```

EXCEPTIONREPORTRECORD

This structure contains machine-independent information about an exception or unwind. No system exception will ever have more parameters than the value of EXCEPTION_MAXIMUM_PARAMETERS. User exceptions are not bound to this limit.

```
typedef STRUCT _EXCEPTIONREPORTRECORD {
  ULONG                              ExceptionNum;                                 /*  Exception number. */
  ULONG                              fHandlerFlags;                                /*  Handler flags. */
  STRUCT _EXCEPTIONREPORTRECORD      *NestedExceptionReportRecord;                 /*  Nested exception report
  PVOID                              ExceptionAddress;                             /*  Address of the exception
  ULONG                              cParameters;                                  /*  Size of exception specif
  ULONG                              ExceptionInfo[EXCEPTION_MAXIMUM_PARAMETERS];  /*  Exception specific infor
} EXCEPTIONREPORTRECORD;

typedef EXCEPTIONREPORTRECORD *PEXCEPTIONREPORTRECORD;
```

EXCEPTIONREPORTRECORD Field - ExceptionNum

ExceptionNum (ULONG)
Exception number.

EXCEPTIONREPORTRECORD Field - fHandlerFlags

fHandlerFlags (ULONG)
Handler flags.

EXCEPTIONREPORTRECORD Field - NestedExceptionReportRec

NestedExceptionReportRecord (STRUCT _EXCEPTIONREPORTRECORD *)
Nested exception report record structure.

EXCEPTIONREPORTRECORD Field - ExceptionAddress

ExceptionAddress (PVOID)
Address of the exception.

EXCEPTIONREPORTRECORD Field - cParameters

cParameters (ULONG)
Size of exception specific information.

EXCEPTIONREPORTRECORD Field - ExceptionInfo[EXCEPTION_MAXIMUM_PARAMETERS]

ExceptionInfo[EXCEPTION_MAXIMUM_PARAMETERS] (ULONG)
Exception specific information.

FATTRS

Font-attributes structure.

```
typedef struct _FATTRS {
  USHORT     usRecordLength;        /*  Length of record. */
  USHORT     fsSelection;           /*  Selection indicators. */
  LONG       lMatch;                /*  Matched-font identity. */
  CHAR       szFacename[FACESIZE];  /*  Typeface name. */
  USHORT     idRegistry;            /*  Registry identifier. */
  USHORT     usCodePage;            /*  Code page. */
  LONG       lMaxBaselineExt;       /*  Maximum baseline extension. */
  LONG       lAveCharWidth;         /*  Average character width. */
  USHORT     fsType;                /*  Type indicators. */
  USHORT     fsFontUse;             /*  Font-use indicators. */
} FATTRS;

typedef FATTRS *PFATTRS;
```

FATTRS Field - usRecordLength

usRecordLength (USHORT)
Length of record.

FATTRS Field - fsSelection

fsSelection (USHORT)
Selection indicators.

Flags causing the following features to be simulated by the system.

Note: If an italic flag is applied to a font that is itself defined as italic, the font is slanted further by italic simulation.

Underscore or strikeout lines are drawn using the appropriate attributes (for example, color) from the character bundle (see the CHARBUNDLE datatype), not the line bundle (see LINEBUNDLE). The width of the line, and the vertical position of the line in font space, are determined by the font. Horizontally, the line starts from a point in font space directly above or below the start point of each character, and extends to a point directly above or below the escapement point for that character.

For this purpose, the start and escapement points are those applicable to left-to-right or right-to-left character directions (see GpiSetCharDirection in *Graphics Programming Interface Programming Reference*), even if the string is currently being drawn in a top-to-bottom or bottom-to-top direction.

For left-to-right or right-to-left directions, any white space generated by the character extra and character break extra attributes (see GpiSetCharExtra and GpiSetCharBreakExtra in *Graphics Programming Interface Programming Reference*), as well as increments provided by the vector of increments on GpiCharStringPos and GpiCharStringPosAt, are also underlined/overstruck, so that in these cases the line is continuous for the string.

FATTR_SEL_ITALIC	Generate *italic* font.
FATTR_SEL_UNDERSCORE	Generate underscored font.
FATTR_SEL_BOLD	Generate **bold** font. (Note that the resulting characters are wider than those in the original font.)
FATTR_SEL_STRIKEOUT	Generate font with overstruck characters.

FATTR_SEL_OUTLINE	Use an outline font with hollow characters. If this flag is not set, outline font characters are filled. Setting this flag normally gives better performance, and for sufficiently small characters (depending on device resolution) there may be little visual difference.

FATTRS Field - lMatch

lMatch (LONG)
Matched-font identity.

FATTRS Field - szFacename[FACESIZE]

szFacename[FACESIZE] (CHAR)
Typeface name.

The typeface name of the font, for example, Tms Rmn.

FATTRS Field - idRegistry

idRegistry (USHORT)
Registry identifier.

Font registry identifier (zero if unknown).

FATTRS Field - usCodePage

usCodePage (USHORT)
Code page.

If zero, the current Gpi code page (see GpiSetCp in *Graphics Programming Interface Programming Reference*) is used. A subsequent GpiSetCp function changes the code page used for this logical font.

FATTRS Field - lMaxBaselineExt

lMaxBaselineExt (LONG)
Maximum baseline extension.

For raster fonts, this should be the height of the required font, in world coordinates.

For outline fonts, this should be zero.

FATTRS Field - lAveCharWidth

lAveCharWidth (LONG)
Average character width.

For raster fonts, this should be the width of the required font, in world coordinates.

For outline fonts, this should be zero.

FATTRS Field - fsType

fsType (USHORT)
Type indicators.

FATTR_TYPE_KERNING	Enable kerning (PostScript only).
FATTR_TYPE_MBCS	Font for mixed single- and double-byte code pages.
FATTR_TYPE_DBCS	Font for double-byte code pages.
FATTR_TYPE_ANTIALIASED	Antialiased font required. Only valid if supported by the device driver.

FATTRS Field - fsFontUse

fsFontUse (USHORT)
Font-use indicators.

These flags indicate how the font is to be used. They affect presentation speed and font quality.

FATTR_FONTUSE_NOMIX	Text is not mixed with graphics and can be written without regard to any interaction with graphics objects.
FATTR_FONTUSE_OUTLINE	Select an outline (vector) font. The font characters can be used as part of a path definition. If this flag is not set, an outline font might or might not be selected. If an outline font is selected, however, character widths are rounded to an integral number of pels.
FATTR_FONTUSE_TRANSFORMABLE	Characters can be transformed (for example, scaled, rotated, or sheared).

FDATE

Date data structure for file-system functions.

```
typedef struct _FDATE {
  USHORT     day:5;    /*  Binary day for directory entry. */
  USHORT     month:4;  /*  Binary month for directory entry. */
  USHORT     year:7;   /*  The number of years since 1980 for this directory entry. */
} FDATE;

typedef FDATE *PFDATE;
```

FDATE Field - day:5

day:5 (USHORT)
Binary day for directory entry.

FDATE Field - month:4

month:4 (USHORT)
Binary month for directory entry.

FDATE Field - year:7

year:7 (USHORT)
The number of years since 1980 for this directory entry.

FEA2

FEA2 defines the format for setting the full extended attributes in the file.

```
typedef struct _FEA2 {
  ULONG      oNextEntryOffset;  /*  Offset to next entry. */
  BYTE       fEA;               /*  Extended attributes flag. */
  BYTE       cbName;            /*  Length of szName, not including NULL. */
  USHORT     cbValue;           /*  Value length. */
  CHAR       szName[1];         /*  Extended attribute name. */
} FEA2;

typedef FEA2 *PFEA2;
```

Extended attributes (EAs) are non-critical by default. A non-critical EA is one that is not necessary to the functionality of the application. If a non-critical EA is lost, the system continues to operate correctly. For example, losing the icons associated with data files does not generally cause any ill effect other than the inability to show the icon.

A *critical extended attribute* is one which is necessary for the correct operation of the operating system or of a particular operation. EAs

should be marked as critical if their loss would cause the system or program to perform incorrectly. For example, a mail program might store mail headers in EAs. The loss of the header from a message would normally render the mail program unable to further use that message. This would be unacceptable, so the mail program should mark this EA as critical.

FEA2 Field - oNextEntryOffset

oNextEntryOffset (ULONG)
Offset to next entry.

FEA2 Field - fEA

fEA (BYTE)
Extended attributes flag.

FEA_NEEDEA
Extended attributes are critical.

If this flag is set, this file cannot be copied to a file system that does not support extended attributes. This flag should only be set if extended attributes are critical to the processing of this file.

0
Extended attributes are not critical.

FEA2 Field - cbName

cbName (BYTE)
Length of *szName* , not including NULL.

This value must be greater than 0.

FEA2 Field - cbValue

cbValue (USHORT)
Value length.

Sending an EA with *cbValue* set to 0 in the FEA2 data structure causes that attribute to be deleted, if possible. Receiving an EA with *cbValue* set to 0 in the FEA2 data structure indicates that the attribute is not present.

FEA2 Field - szName[1]

szName[1] (CHAR)

Extended attribute name.

FEA2LIST

FEA2 data structure.

```
typedef struct _FEA2LIST {
  ULONG     cbList;   /*  Total bytes of structure including full list. */
  FEA2      list[1];  /*  Variable-length FEA2 structures. */
} FEA2LIST;

typedef FEA2LIST *PFEA2LIST;
```

FEA2LIST Field - cbList

cbList (ULONG)
Total bytes of structure including full list.

FEA2LIST Field - list[1]

list[1] (FEA2)
Variable-length FEA2 structures.

FHLOCK

Unsigned integer in the range 0 through 4 294 967 295.

```
typedef ULONG FHLOCK;
```

FILEFINDBUF

Find file buffer data structure.

```
typedef struct _FILEFINDBUF {
  FDATE      fdateCreation;           /*  Date of file creation. */
  FTIME      ftimeCreation;           /*  Time of file creation. */
  FDATE      fdateLastAccess;         /*  Date of last access. */
  FTIME      ftimeLastAccess;         /*  Time of last access. */
  FDATE      fdateLastWrite;          /*  Date of last write. */
  FTIME      ftimeLastWrite;          /*  Time of last write. */
  ULONG      cbFile;                  /*  Size of file. */
  ULONG      cbFileAlloc;             /*  Allocated size. */
```

```
  USHORT     attrFile;               /*  File attributes. */
  UCHAR      cchName;                /*  Length of file name. */
  CHAR       achName[CCHMAXPATHCOMP];  /*  File name including null terminator. */
} FILEFINDBUF;

typedef FILEFINDBUF *PFILEFINDBUF;
```

FILEFINDBUF Field - fdateCreation

fdateCreation (FDATE)
Date of file creation.

FILEFINDBUF Field - ftimeCreation

ftimeCreation (FTIME)
Time of file creation.

FILEFINDBUF Field - fdateLastAccess

fdateLastAccess (FDATE)
Date of last access.

FILEFINDBUF Field - ftimeLastAccess

ftimeLastAccess (FTIME)
Time of last access.

FILEFINDBUF Field - fdateLastWrite

fdateLastWrite (FDATE)
Date of last write.

FILEFINDBUF Field - ftimeLastWrite

ftimeLastWrite (FTIME)
Time of last write.

FILEFINDBUF Field - cbFile

cbFile (ULONG)
Size of file.

FILEFINDBUF Field - cbFileAlloc

cbFileAlloc (ULONG)
Allocated size.

FILEFINDBUF Field - attrFile

attrFile (USHORT)
File attributes.

FILEFINDBUF Field - cchName

cchName (UCHAR)
Length of file name.

FILEFINDBUF Field - achName[CCHMAXPATHCOMP]

achName[CCHMAXPATHCOMP] (CHAR)
File name including null terminator.

FILEFINDBUF3

Level 1 (32-bit) information (used without EAs).

```
typedef struct _FILEFINDBUF3 {
  ULONG     oNextEntryOffset;         /*  Offset of next entry. */
  FDATE     fdateCreation;            /*  Date of file creation. */
  FTIME     ftimeCreation;            /*  Time of file creation. */
  FDATE     fdateLastAccess;          /*  Date of last access. */
  FTIME     ftimeLastAccess;          /*  Time of last access. */
  FDATE     fdateLastWrite;           /*  Date of last write. */
  FTIME     ftimeLastWrite;           /*  Time of last write. */
  ULONG     cbFile;                   /*  Size of file. */
  ULONG     cbFileAlloc;              /*  Allocation size. */
  ULONG     attrFile;                 /*  File attributes. */
  UCHAR     cchName;
  CHAR      achName[CCHMAXPATHCOMP];  /*  File name including null terminator. */
} FILEFINDBUF3;

typedef FILEFINDBUF3 *PFILEFINDBUF3;
```

FILEFINDBUF3 Field - oNextEntryOffset

oNextEntryOffset (ULONG)
Offset of next entry.

FILEFINDBUF3 Field - fdateCreation

fdateCreation (FDATE)
Date of file creation.

FILEFINDBUF3 Field - ftimeCreation

ftimeCreation (FTIME)
Time of file creation.

FILEFINDBUF3 Field - fdateLastAccess

fdateLastAccess (FDATE)
Date of last access.

FILEFINDBUF3 Field - ftimeLastAccess

ftimeLastAccess (FTIME)

Time of last access.

FILEFINDBUF3 Field - fdateLastWrite

fdateLastWrite (FDATE)
Date of last write.

FILEFINDBUF3 Field - ftimeLastWrite

ftimeLastWrite (FTIME)
Time of last write.

FILEFINDBUF3 Field - cbFile

cbFile (ULONG)
Size of file.

FILEFINDBUF3 Field - cbFileAlloc

cbFileAlloc (ULONG)
Allocation size.

FILEFINDBUF3 Field - attrFile

attrFile (ULONG)
File attributes.

FILEFINDBUF3 Field - cchName

cchName (UCHAR)

FILEFINDBUF3 Field - achName[CCHMAXPATHCOMP]

achName[CCHMAXPATHCOMP] (CHAR)
File name including null terminator.

FILEFINDBUF4

Level 2 (32-bit) information (used with EAs).

```
typedef struct _FILEFINDBUF4 {
  ULONG     oNextEntryOffset;         /*  Offset of next entry. */
  FDATE     fdateCreation;            /*  Date of file creation. */
  FTIME     ftimeCreation;            /*  Time of file creation. */
  FDATE     fdateLastAccess;          /*  Date of last access. */
  FTIME     ftimeLastAccess;          /*  Time of last access. */
  FDATE     fdateLastWrite;           /*  Date of last write. */
  FTIME     ftimeLastWrite;           /*  Time of last write. */
  ULONG     cbFile;                   /*  Size of file. */
  ULONG     cbFileAlloc;              /*  Allocated size. */
  ULONG     attrFile;                 /*  File attributes. */
  ULONG     cbList;                   /*  Size of the file's extended attributes. */
  UCHAR     cchName;                  /*  Length of file name. */
  CHAR      achName[CCHMAXPATHCOMP];  /*  File name including null terminator. */
} FILEFINDBUF4;

typedef FILEFINDBUF4 *PFILEFINDBUF4;
```

FILEFINDBUF4 Field - oNextEntryOffset

oNextEntryOffset (ULONG)
Offset of next entry.

FILEFINDBUF4 Field - fdateCreation

fdateCreation (FDATE)
Date of file creation.

FILEFINDBUF4 Field - ftimeCreation

ftimeCreation (FTIME)
Time of file creation.

FILEFINDBUF4 Field - fdateLastAccess

fdateLastAccess (FDATE)
Date of last access.

FILEFINDBUF4 Field - ftimeLastAccess

ftimeLastAccess (FTIME)
Time of last access.

FILEFINDBUF4 Field - fdateLastWrite

fdateLastWrite (FDATE)
Date of last write.

FILEFINDBUF4 Field - ftimeLastWrite

ftimeLastWrite (FTIME)
Time of last write.

FILEFINDBUF4 Field - cbFile

cbFile (ULONG)
Size of file.

FILEFINDBUF4 Field - cbFileAlloc

cbFileAlloc (ULONG)
Allocated size.

FILEFINDBUF4 Field - attrFile

attrFile (ULONG)
File attributes.

FILEFINDBUF4 Field - cbList

cbList (ULONG)
Size of the file's extended attributes.

The size is measured in bytes and is the size of the file's entire extended attribute set on the disk.

FILEFINDBUF4 Field - cchName

cchName (UCHAR)
Length of file name.

FILEFINDBUF4 Field - achName[CCHMAXPATHCOMP]

achName[CCHMAXPATHCOMP] (CHAR)
File name including null terminator.

FILELOCK

FILELOCK data structure.

```
typedef struct _FILELOCK {
  LONG     lOffset;  /*  Offset to the beginning of the lock (or unlock) range. */
  LONG     lRange;   /*  Length, in bytes, of the lock (or unlock) range. */
} FILELOCK;

typedef FILELOCK *PFILELOCK;
```

FILELOCK Field - lOffset

lOffset (LONG)
Offset to the beginning of the lock (or unlock) range.

FILELOCK Field - lRange

lRange (LONG)
Length, in bytes, of the lock (or unlock) range.

A value of 0 indicates that locking (or unlocking) is not required.

FILESTATUS

Use for Level 1 (FIL_STANDARD) file information for:

- DosQueryFileInfo
- DosQueryPathInfo
- DosSetFileInfo
- DosSetPathInfo

```
typedef struct _FILESTATUS {
  FDATE      fdateCreation;    /*  Date of file creation. */
  FTIME      ftimeCreation;    /*  Time of file creation. */
  FDATE      fdateLastAccess;  /*  Date of last access. */
  FTIME      ftimeLastAccess;  /*  Time of last access. */
  FDATE      fdateLastWrite;   /*  Date of last write. */
  FTIME      ftimeLastWrite;   /*  Time of last write. */
  ULONG      cbFile;           /*  File size (end of data). */
  ULONG      cbFileAlloc;      /*  File allocated size. */
  USHORT     attrFile;         /*  Attributes of the file. */
} FILESTATUS;

typedef FILESTATUS *PFILESTATUS;
```

FILESTATUS Field - fdateCreation

fdateCreation (FDATE)
Date of file creation.

FILESTATUS Field - ftimeCreation

ftimeCreation (FTIME)
Time of file creation.

FILESTATUS Field - fdateLastAccess

fdateLastAccess (FDATE)
Date of last access.

FILESTATUS Field - ftimeLastAccess

ftimeLastAccess (FTIME)
Time of last access.

FILESTATUS Field - fdateLastWrite

fdateLastWrite (FDATE)
Date of last write.

FILESTATUS Field - ftimeLastWrite

ftimeLastWrite (FTIME)
Time of last write.

FILESTATUS Field - cbFile

cbFile (ULONG)
File size (end of data).

FILESTATUS Field - cbFileAlloc

cbFileAlloc (ULONG)
File allocated size.

FILESTATUS Field - attrFile

attrFile (USHORT)
Attributes of the file.

FILESTATUS3

Level 1 (32-bit) (FIL_STANDARD) information.

```
typedef struct _FILESTATUS3 {
  FDATE     fdateCreation;    /*  Date of file creation. */
  FTIME     ftimeCreation;    /*  Time of file creation. */
  FDATE     fdateLastAccess;  /*  Date of last access. */
  FTIME     ftimeLastAccess;  /*  Time of last access. */
  FDATE     fdateLastWrite;   /*  Date of last write. */
  FTIME     ftimeLastWrite;   /*  Time of last write. */
  ULONG     cbFile;           /*  File size (end of data). */
  ULONG     cbFileAlloc;      /*  File allocated size. */
  ULONG     attrFile;         /*  Attributes of the file. */
} FILESTATUS3;

typedef FILESTATUS3 *PFILESTATUS3;
```

FILESTATUS3 Field - fdateCreation

fdateCreation (FDATE)
Date of file creation.

FILESTATUS3 Field - ftimeCreation

ftimeCreation (FTIME)
Time of file creation.

FILESTATUS3 Field - fdateLastAccess

fdateLastAccess (FDATE)
Date of last access.

FILESTATUS3 Field - ftimeLastAccess

ftimeLastAccess (FTIME)
Time of last access.

FILESTATUS3 Field - fdateLastWrite

fdateLastWrite (FDATE)
Date of last write.

FILESTATUS3 Field - ftimeLastWrite

ftimeLastWrite (FTIME)
Time of last write.

FILESTATUS3 Field - cbFile

cbFile (ULONG)
File size (end of data).

FILESTATUS3 Field - cbFileAlloc

cbFileAlloc (ULONG)
File allocated size.

FILESTATUS3 Field - attrFile

attrFile (ULONG)
Attributes of the file.

FILESTATUS4

Level 2 (32-bit) (FIL_QUERYEASIZE) information.

```
typedef struct _FILESTATUS4 {
  FDATE     fdateCreation;    /*  Date of file creation. */
  FTIME     ftimeCreation;    /*  Time of file creation. */
  FDATE     fdateLastAccess;  /*  Date of last access. */
  FTIME     ftimeLastAccess;  /*  Time of last access. */
  FDATE     fdateLastWrite;   /*  Date of last write. */
  FTIME     ftimeLastWrite;   /*  Time of last write. */
  ULONG     cbFile;           /*  File size (end of data). */
  ULONG     cbFileAlloc;      /*  File allocated size. */
  ULONG     attrFile;         /*  Attributes of the file. */
  ULONG     cbList;           /*  Length of entire EA set. */
} FILESTATUS4;
```

```
typedef FILESTATUS4 *PFILESTATUS4;
```

FILESTATUS4 Field - fdateCreation

fdateCreation (FDATE)
Date of file creation.

FILESTATUS4 Field - ftimeCreation

ftimeCreation (FTIME)
Time of file creation.

FILESTATUS4 Field - fdateLastAccess

fdateLastAccess (FDATE)
Date of last access.

FILESTATUS4 Field - ftimeLastAccess

ftimeLastAccess (FTIME)
Time of last access.

FILESTATUS4 Field - fdateLastWrite

fdateLastWrite (FDATE)
Date of last write.

FILESTATUS4 Field - ftimeLastWrite

ftimeLastWrite (FTIME)
Time of last write.

FILESTATUS4 Field - cbFile

cbFile (ULONG)
File size (end of data).

FILESTATUS4 Field - cbFileAlloc

cbFileAlloc (ULONG)
File allocated size.

FILESTATUS4 Field - attrFile

attrFile (ULONG)
Attributes of the file.

FILESTATUS4 Field - cbList

cbList (ULONG)
Length of entire EA set.

FONTMETRICS

Font-metrics structure.

This structure is returned to applications on the GpiQueryFonts and GpiQueryFontMetrics calls and conveys information from the font creator to the application.

```
typedef struct _FONTMETRICS {
  CHAR       szFamilyname[FACESIZE];  /*  Family name. */
  CHAR       szFacename[FACESIZE];    /*  Face name. */
  USHORT     idRegistry;              /*  Registry identifier. */
  USHORT     usCodePage;              /*  Code page. */
  LONG       lEmHeight;               /*  Em height. */
  LONG       lXHeight;                /*  X height. */
  LONG       lMaxAscender;            /*  Maximum ascender. */
  LONG       lMaxDescender;           /*  Maximum descender. */
  LONG       lLowerCaseAscent;        /*  Lowercase ascent. */
  LONG       lLowerCaseDescent;       /*  Lowercase descent. */
  LONG       lInternalLeading;        /*  Internal leading. */
  LONG       lExternalLeading;        /*  External leading. */
  LONG       lAveCharWidth;           /*  Average character width. */
```

```
  LONG        lMaxCharInc;            /*  Maximum character increment. */
  LONG        lEmInc;                 /*  Em increment. */
  LONG        lMaxBaselineExt;        /*  Maximum baseline extent. */
  SHORT       sCharSlope;             /*  Character slope. */
  SHORT       sInlineDir;             /*  Inline direction. */
  SHORT       sCharRot;               /*  Character rotation. */
  USHORT      usWeightClass;          /*  Weight class. */
  USHORT      usWidthClass;           /*  Width class. */
  SHORT       sXDeviceRes;            /*  X-device resolution. */
  SHORT       sYDeviceRes;            /*  Y-device resolution. */
  SHORT       sFirstChar;             /*  First character. */
  SHORT       sLastChar;              /*  Last character. */
  SHORT       sDefaultChar;           /*  Default character. */
  SHORT       sBreakChar;             /*  Break character. */
  SHORT       sNominalPointSize;      /*  Nominal point size. */
  SHORT       sMinimumPointSize;      /*  Minimum point size. */
  SHORT       sMaximumPointSize;      /*  Maximum point size. */
  USHORT      fsType;                 /*  Type indicators. */
  USHORT      fsDefn;                 /*  Definition indicators. */
  USHORT      fsSelection;            /*  Selection indicators. */
  USHORT      fsCapabilities;         /*  Font capabilities. */
  LONG        lSubscriptXSize;        /*  Subscript x-size. */
  LONG        lSubscriptYSize;        /*  Subscript y-size. */
  LONG        lSubscriptXOffset;      /*  Subscript x-offset. */
  LONG        lSubscriptYOffset;      /*  Subscript y-offset. */
  LONG        lSuperscriptXSize;      /*  Superscript x-size. */
  LONG        lSuperscriptYSize;      /*  Superscript y-size. */
  LONG        lSuperscriptXOffset;    /*  Superscript x-offset. */
  LONG        lSuperscriptYOffset;    /*  Superscript y-offset. */
  LONG        lUnderscoreSize;        /*  Underscore size. */
  LONG        lUnderscorePosition;    /*  Underscore position. */
  LONG        lStrikeoutSize;         /*  Strikeout size. */
  LONG        lStrikeoutPosition;     /*  Strikeout position. */
  SHORT       sKerningPairs;          /*  Kerning pairs. */
  SHORT       sFamilyClass;           /*  Font family design classification. */
  LONG        lMatch;                 /*  Matched font identity. */
  LONG        FamilyNameAtom;         /*  Font family name atom. */
  LONG        FaceNameAtom;           /*  Font facename atom. */
  PANOSE      panose;                 /*  Panose font descriptor. */
} FONTMETRICS;

typedef FONTMETRICS *PFONTMETRICS;
```

FONTMETRICS Field - szFamilyname[FACESIZE]

szFamilyname[FACESIZE] (CHAR)
Family name.

The family name of the font that describes the basic appearance of the font, for example, Times New Roman** This string is null terminated, and therefore is limited to 31 characters in length. Longer names may be retrieved by using the *FamilyNameAtom* field to retrieve the full name from the System Atom table.

FONTMETRICS Field - szFacename[FACESIZE]

szFacename[FACESIZE] (CHAR)
Face name.

The typeface name that defines the particular font, for example, Times New Roman Bold Italic. This string is null terminated, and therefore is limited to 31 characters in length. Longer names may be retrieved by using the *FaceNameAtom* field to retrieve the full name from the System Atom table.

FONTMETRICS Field - idRegistry

idRegistry (USHORT)
Registry identifier.

The IBM registered number (or zero).

FONTMETRICS Field - usCodePage

usCodePage (USHORT)
Code page.

Defines the registered code page supported by the font. For example, the original IBM PC code page is 437. A value of 0 implies that the font may be used with any of the OS/2 supported code pages.

Where a font contains special symbols for which there is no registered code page, then code page 65400 is used.

FONTMETRICS Field - lEmHeight

lEmHeight (LONG)
Em height.

The height of the Em square in world coordinate units. This corresponds to the point size for the font.

FONTMETRICS Field - lXHeight

lXHeight (LONG)
X height.

The nominal height above the baseline for lowercase characters (ignoring ascenders) in world coordinate units.

FONTMETRICS Field - lMaxAscender

lMaxAscender (LONG)
Maximum ascender.

The maximum height above the baseline reached by any part of any symbol in the font in world coordinate units. This field may exceed *lEmHeight* .

FONTMETRICS Field - lMaxDescender

lMaxDescender (LONG)
Maximum descender.

The maximum depth below the baseline reached by any part of any symbol in the font in world coordinate units. This field may exceed *lEmHeight* .

FONTMETRICS Field - lLowerCaseAscent

lLowerCaseAscent (LONG)
Lowercase ascent.

The maximum height above the baseline reached by any part of any lowercase (Latin unaccented "a" through "z") symbol in the font in world coordinate units.

FONTMETRICS Field - lLowerCaseDescent

lLowerCaseDescent (LONG)
Lowercase descent.

The maximum depth below the baseline reached by any part of any lowercase (Latin unaccented "a" through "z") symbol in the font in world coordinate units.

FONTMETRICS Field - lInternalLeading

lInternalLeading (LONG)
Internal leading.

The amount of space which, when subtracted from *lMaxAscender* , gives a font-design dependent, but glyph-set independent, measure of the distance above the baseline that characters extend. This calculation approximates the visual top to a row of characters without actually looking at the characters in the row.

For optimum results, this field should be used by applications to position the first line of a block of text by subtracting it from *lMaxAscender* and positioning the baseline that distance below whatever is above the text.

Note: This does not guarantee that characters will not overwrite information above them, but does give a font designer's view of where to place the text. Collision should be tested for, and additional space allocated if necessary.

FONTMETRICS Field - lExternalLeading

lExternalLeading (LONG)
External leading.

The amount of guaranteed white space advised by the font designer to appear between adjacent rows of text. This value may be zero.

Note: The fonts built in to Presentation Manager have zero in this field.

FONTMETRICS Field - lAveCharWidth

lAveCharWidth (LONG)
Average character width.

This is determined by multiplying the width of each lowercase character by a constant, adding the products, and then dividing by 1000. The letters involved in this, plus their constants, are as follows:

Letter	Constant
a	64
b	14
c	27
d	35
e	100
f	20
g	14
h	42
i	63
j	3
k	6
l	35
m	20
n	56
o	56
p	17
q	4
r	49
s	56
t	71
u	31
v	10
w	18
x	3
y	18
z	2
space	166

Note: For fixed pitch fonts, this value will be the same as the (A width + B width + C width) escapement of each character.

FONTMETRICS Field - lMaxCharInc

lMaxCharInc (LONG)
Maximum character increment.

The maximum character increment for the font in world coordinate units.

FONTMETRICS Field - lEmInc

lEmInc (LONG)
Em increment.

The width of the Em square in world coordinate units. This corresponds to the point size of the font. When the horizontal device resolution equals the vertical device resolution this is equal to the em height.

FONTMETRICS Field - lMaxBaselineExt

lMaxBaselineExt (LONG)
Maximum baseline extent.

The maximum vertical space occupied by the font, in world coordinate units. This is the sum of *lMaxAscender* and *lMaxDescender* if both are positive. It is also the sum of *lInternalLeading* and *lEmHeight*.

One possible type of line spacing can be computed by adding *lMaxBaselineExt* to *lExternalLeading*. Such a line spacing, however, would be dependent on the glyph set included in the font. If a new version of the font should be made available, with new glyphs, then it is possible that this value will change because one of the new glyphs has gone above the previous *lMaxAscender* or below the previous *lMaxDescender*. More sophisticated applications will base line spacing on the point size (*lEmHeight*) of the font, which is an invariant of the font, multiplied by some factor (for example, 120%) plus any external leading.

This field may exceed *lEmHeight*.

FONTMETRICS Field - sCharSlope

sCharSlope (SHORT)
Character slope.

Defines the nominal slope for the characters of a font. The slope is defined in degrees increasing clockwise from the vertical. An *italic* font is an example of a font with a nonzero slope.

Note: The units for this metric are degrees and minutes, encoded as shown in the following example:

```
180 degrees 59 minutes would be represented as :

    < byte 1  >     < byte 2   >

    < Minutes >     < Degrees >

 0 1 1 1 0 1 1   0 1 0 1 1 0 1 0 0

     59 min         180 degrees
```

FONTMETRICS Field - sInlineDir

sInlineDir (SHORT)
Inline direction.

The direction in which the characters in the font are designed for viewing. The direction is defined in degrees increasing clockwise from the horizontal (left-to-right). Characters are added to a line of text in the inline direction.

Note: The units for this metric are degrees and minutes, encoded as shown in *sCharSlope*.

FONTMETRICS Field - sCharRot

sCharRot (SHORT)
Character rotation.

The rotation of the character glyphs with respect to the baseline, the angle increasing counter clockwise. This is the angle assigned by the font designer.

Note: The units for this metric are degrees and minutes, encoded as shown in *sCharSlope* .

FONTMETRICS Field - usWeightClass

usWeightClass (USHORT)
Weight class.

Indicates the visual weight (thickness of strokes) of the characters in the font:

Value	Description
1	Ultra-light
2	Extra-light
3	Light
4	Semi-light
5	Medium (normal)
6	Semi-bold
7	Bold
8	Extra-bold
9	Ultra-bold

FONTMETRICS Field - usWidthClass

usWidthClass (USHORT)
Width class.

Indicates the relative aspect ratio of the characters of the font in relation to the normal aspect ratio for this type of font:

```
Value Description     % of normal width
1     Ultra-condensed 50
2     Extra-condensed 62.5
3     Condensed       75
4     Semi-condensed  87.5
5     Medium (normal) 100
6     Semi-expanded   112.5
7     Expanded        125
8     Extra-expanded  150
9     Ultra-expanded  200
```

FONTMETRICS Field - sXDeviceRes

sXDeviceRes (SHORT)
X-device resolution.

For bit-map fonts this is the resolution in the X direction of the intended target device, measured in pels per inch.

For outline fonts this is the number of notional units in the X direction of the Em square, measured in notional units per Em. (Notional units are the units in which the outline is defined.)

FONTMETRICS Field - sYDeviceRes

sYDeviceRes (SHORT)
Y-device resolution.

For bit-map fonts this is the resolution in the Y direction of the intended target device, measured in pels per inch.

For outline fonts this is the number of notional units in the Y direction of the Em square, measured in notional units per Em. (Notional units are the units in which the outline is defined.)

FONTMETRICS Field - sFirstChar

sFirstChar (SHORT)
First character.

The code point of the first character in the font.

FONTMETRICS Field - sLastChar

sLastChar (SHORT)
Last character.

The code point of the last character in the font, expressed as an offset from *sFirstChar* .

All code points between the first and last character specified must be supported by the font.

FONTMETRICS Field - sDefaultChar

sDefaultChar (SHORT)
Default character.

The code point that is used if a code point outside the range supported by the font is used, expressed as an offset from *sFirstChar* .

FONTMETRICS Field - sBreakChar

sBreakChar (SHORT)
Break character.

The code point that represents the "space" or "break" character for this font, expressed as an offset from *sFirstChar* . For example, if the first character is the space in code page 850, *sFirstChar* = 32, and *sBreakChar* = 0.

FONTMETRICS Field - sNominalPointSize

sNominalPointSize (SHORT)
Nominal point size.

For a bit-map font, this field contains the height of the font.

For an outline font, this field contains the height intended by the font designer. For example, some fonts are designed for text use in which case a value of 120 (12 point) would probably be placed in this field, whereas other fonts are designed for "display" use ("display" is typographer's terminology for larger sizes). This is not the only size at which the font can be used.

Measured in decipoints (a decipoint is 1/720th of an inch).

FONTMETRICS Field - sMinimumPointSize

sMinimumPointSize (SHORT)
Minimum point size.

For a bit-map font, this field does not apply. For an outline font, this field contains the minimum height intended by the font designer. Note that this is not a restriction of the size at which the font can be used.

Measured in decipoints (a decipoint is 1/720th of an inch).

FONTMETRICS Field - sMaximumPointSize

sMaximumPointSize (SHORT)
Maximum point size.

For a bit-map font, this field does not apply.

For an outline font, this field contains the maximum height intended by the font designer. Note that this is not a restriction of the size at which the font can be used.

Measured in decipoints (a decipoint is 1/720th of an inch).

FONTMETRICS Field - fsType

fsType (USHORT)
Type indicators.

This field contains the following information:

FM_TYPE_FIXED	Characters in the font have the same fixed width.
FM_TYPE_LICENSED	Licensed (protected) font.
FM_TYPE_KERNING	Font contains kerning information.
FM_TYPE_64K	Font is larger than 64KB (KB equals 1024 bytes) in size. If the following two bits are false, the font is for single-byte code pages. One of the bits may be set.
FM_TYPE_DBCS	Font is for double-byte code pages.
FM_TYPE_MBCS	Font is for mixed single- or double-byte code pages.

FM_TYPE_FACETRUNC
Font *szFacename* has been truncated.
FM_TYPE_FAMTRUNC
Font *szFamilyname* has been truncated.
FM_TYPE_ATOMS
The System Atom table atom values in *FamilyNameAtom* and in *FaceNameAtom* are valid.

FONTMETRICS Field - fsDefn

fsDefn (USHORT)
Definition indicators.

Contains the following font definition data:

FM_DEFN_OUTLINE	Font is a vector (outline) font; otherwise, it is a bit-map font.
FM_DEFN_GENERIC	Font is in a format that can be used by the GPI; otherwise, it is a device font.

FONTMETRICS Field - fsSelection

fsSelection (USHORT)
Selection indicators.

Contains information about the font patterns in the physical font.

Note: The flags do not reflect simulations applied to the physical font.

Possible values are:

FM_SEL_ITALIC	True indicates that this font is designed as an italic font.
FM_SEL_UNDERSCORE	TRUE indicates that this font is designed with underscores included in each character.
FM_SEL_NEGATIVE	TRUE indicates that this font is designed with the background and foreground reversed.
FM_SEL_OUTLINE	TRUE indicates that this font is designed with outline (hollow) characters.

FM_SEL_STRIKEOUT	TRUE indicates that this font is designed with an overstrike through each character.
FM_SEL_BOLD	TRUE indicates that this font is designed with bold characters.
FM_SEL_ISO9241_TESTED	This flag indicates that the font has been tested for compliance to ISO 9241. The presence of this flag doesn't indicate whether the font passed or failed, only that it was tested. **Note:** While the fonts were primarily tested for meeting the ISO standard, they have also been designed to meet the German standard DIN 66 234. Where the two standards differ, the fonts have been designed to meet the more stringent requirement.

FONTMETRICS Field - fsCapabilities

fsCapabilities (USHORT)
Font capabilities.

This attribute applies only to device fonts.

FM_CAP_NOMIX	Characters may not be mixed with graphics.
QUALITY	The most significant byte may contain the following numeric value: 0 Undefined 1 DP quality 2 DP draft 3 Near Letter Quality 4 Letter Quality

FONTMETRICS Field - lSubscriptXSize

lSubscriptXSize (LONG)
Subscript x-size.

The horizontal size recommended by the font designer for subscripts for this font in world coordinate units.

FONTMETRICS Field - lSubscriptYSize

lSubscriptYSize (LONG)
Subscript y-size.

The vertical size recommended by the font designer for subscripts for this font in world coordinate units.

FONTMETRICS Field - lSubscriptXOffset

lSubscriptXOffset (LONG)
Subscript x-offset.

The baseline x-offset recommended by the font designer for subscripts for this font in world coordinate units.

FONTMETRICS Field - lSubscriptYOffset

lSubscriptYOffset (LONG)
Subscript y-offset.

The baseline y-offset recommended by the font designer for subscripts for this font in world coordinate units.

Note: Positive numbers indicate an offset below the baseline.

FONTMETRICS Field - lSuperscriptXSize

lSuperscriptXSize (LONG)
Superscript x-size.

The horizontal size recommended by the font designer for superscripts for this font in world coordinate units.

FONTMETRICS Field - lSuperscriptYSize

lSuperscriptYSize (LONG)
Superscript y-size.

The vertical point size recommended by the font designer for superscripts for this font in world coordinate units.

FONTMETRICS Field - lSuperscriptXOffset

lSuperscriptXOffset (LONG)
Superscript x-offset.

The baseline x-offset recommended by the font designer for superscripts for this font in world coordinate units.

FONTMETRICS Field - lSuperscriptYOffset

lSuperscriptYOffset (LONG)
Superscript y-offset.

The baseline y-offset recommended by the font designer for superscripts for this font in world coordinate units.

FONTMETRICS Field - lUnderscoreSize

lUnderscoreSize (LONG)
Underscore size.

The width (thickness) of the underscore stroke in world coordinate units. This describes the actual underscore in the font if FM_SEL_UNDERSCORE is also set. Otherwise it describes what the engine will simulate if underscore is requested in GpiCreateLogFont.

FONTMETRICS Field - lUnderscorePosition

lUnderscorePosition (LONG)
Underscore position.

The position of the underscore stroke from the baseline in world coordinate units. This describes the actual underscore in the font if FM_SEL_UNDERSCORE is also set. Otherwise it describes what the engine will simulate if underscore is requested in GpiCreateLogFont.

Note: Positive values indicate an offset below the baseline.

FONTMETRICS Field - lStrikeoutSize

lStrikeoutSize (LONG)
Strikeout size.

The width of the strikeout stroke in world coordinate units. This describes the actual underscore in the font if FM_SEL_STRIKEOUT is also set. Otherwise it describes what the engine will simulate if overstrike is requested in GpiCreateLogFont.

FONTMETRICS Field - lStrikeoutPosition

lStrikeoutPosition (LONG)
Strikeout position.

The position of the strikeout stroke relative to the baseline in world coordinate units. This describes the actual underscore in the font if FM_SEL_STRIKEOUT is also set. Otherwise it describes what the engine will simulate if overstrike is requested in GpiCreateLogFont.

FONTMETRICS Field - sKerningPairs

sKerningPairs (SHORT)
Kerning pairs.

The number of kerning pairs in the kerning pair table.

FONTMETRICS Field - sFamilyClass

sFamilyClass (SHORT)
Font family design classification.

This value contains a font class and its subclass.

FONTMETRICS Field - lMatch

lMatch (LONG)
Matched font identity.

This uniquely identifies the font for a given device and device driver combination. A positive match number signifies that the font is a generic (engine) font while a negative number indicates a device font (a native or downloadable font). This value should not be used to identify a font across system boundaries.

FONTMETRICS Field - FamilyNameAtom

FamilyNameAtom (LONG)
Font family name atom.

This value contains the atom identifier for the font family name in the System Atom Table.

FONTMETRICS Field - FaceNameAtom

FaceNameAtom (LONG)
Font facename atom.

This value contains the atom identifier for the font face name in the System Atom Table.

FONTMETRICS Field - panose

panose (PANOSE)
Panose font descriptor.

This is the Panose descriptor identifying the visual characteristics of the font.

FPREG

Coprocessor stack register element.

```
typedef struct _FPREG {
  ULONG      losig;    /*  Low 32-bits of the significand. */
  ULONG      hisig;    /*  High 32-bits of the significand. */
  USHORT     signexp;  /*  Sign and exponent. */
} FPREG;

typedef FPREG *PFPREG;
```

A floating point register is 80 bits wide and consists of three fields. The following graphic shows the layout of the floating point register:

```
  79  78         64 63                                    0

                                Significand


             Exponent
   Sign
```

FPREG Field - losig

losig (ULONG)
Low 32-bits of the significand.

The low 32-bits of the number's significant digits are held in the lower part of the significand field.

FPREG Field - hisig

hisig (ULONG)
High 32-bits of the significand.

The high 32-bits of the number's significant digits are held in the higher part of the significand field.

FPREG Field - signexp

signexp (USHORT)
Sign and exponent.

The exponent field (bits 64-78) locates the binary point within the significand field (bits 0-63).

The 1-bit sign field (bit 79) indicates whether the number is positive or negative

FRAME

Frame control data structure.

```
typedef struct _FRAME {
  BYTE     bCharsPerLine;  /*  Characters per line. */
  BYTE     bLinesPerInch;  /*  Lines per inch. */
} FRAME;

typedef FRAME *PFRAME;
```

FRAME Field - bCharsPerLine

bCharsPerLine (BYTE)
Characters per line.

Valid values are 80 and 132.

FRAME Field - bLinesPerInch

bLinesPerInch (BYTE)
Lines per inch.

Valid values are 6 and 8.

FSALLOCATE

File-system device allocation.

```
typedef struct _FSALLOCATE {
  ULONG      idFileSystem;  /*  File system identification. */
  ULONG      cSectorUnit;   /*  Number of sectors per allocation unit. */
  ULONG      cUnit;         /*  Number of allocation units. */
  ULONG      cUnitAvail;    /*  Number of allocation units available. */
  USHORT     cbSector;      /*  Number of bytes per sector. */
} FSALLOCATE;

typedef FSALLOCATE *PFSALLOCATE;
```

FSALLOCATE Field - idFileSystem

idFileSystem (ULONG)
File system identification.

FSALLOCATE Field - cSectorUnit

cSectorUnit (ULONG)
Number of sectors per allocation unit.

FSALLOCATE Field - cUnit

cUnit (ULONG)
Number of allocation units.

FSALLOCATE Field - cUnitAvail

cUnitAvail (ULONG)
Number of allocation units available.

FSALLOCATE Field - cbSector

cbSector (USHORT)
Number of bytes per sector.

FSINFO

File-system information data structure.

```
typedef struct _FSINFO {
  FDATE           fdateCreation;  /*  Creation date. */
  FTIME           ftimeCreation;  /*  Creation time. */
  VOLUMELABEL     vol;    /*  Volume label. */
} FSINFO;

typedef FSINFO *PFSINFO;
```

FSINFO Field - fdateCreation

fdateCreation (FDATE)
Creation date.

Returns the date that the drive was created (formatted).

FSINFO Field - ftimeCreation

ftimeCreation (FTIME)
Creation time.

Returns the time that the drive was created (formatted).

FSINFO Field - vol

vol (VOLUMELABEL)
Volume label.

Trailing blanks are not considered part of the volume label and are not returned. The volume label is limited to a length of 11 bytes.

FSQBUFFER2

Data structure for information about an attached file system (local or remote), or about a character device or pseudocharacter device attached to the file system.

```
typedef struct _FSQBUFFER2 {
  USHORT     iType;         /*  Type of item. */
  USHORT     cbName;        /*  Length, in bytes, of the item name, not including null. */
  USHORT     cbFSDName;     /*  Length, in bytes, of the file-system driver name, not including null. */
  USHORT     cbFSAData;     /*  Length, in bytes, of the file-system driver Attach data returned by the file-sys
  UCHAR      szName[1];     /*  Item name. */
  UCHAR      szFSDName[1];  /*  Name of the file-system driver that the item is attached to. */
  UCHAR      rgFSAData[1];  /*  File-system driver Attach data returned by the file-system driver. */
} FSQBUFFER2;

typedef FSQBUFFER2 *PFSQBUFFER2;
```

FSQBUFFER2 Field - iType

iType (USHORT)
Type of item.

Possible values are described in the following list:

1	FSAT_CHARDEV Resident character device
2	FSAT_PSEUDODEV Pseudocharacter device
3	FSAT_LOCALDRV Local drive
4	FSAT_REMOTEDRV Remote drive attached to the file-system driver.

FSQBUFFER2 Field - cbName

cbName (USHORT)
Length, in bytes, of the item name, not including null.

FSQBUFFER2 Field - cbFSDName

cbFSDName (USHORT)
Length, in bytes, of the file-system driver name, not including null.

FSQBUFFER2 Field - cbFSAData

cbFSAData (USHORT)
Length, in bytes, of the file-system driver Attach data returned by the file-system driver.

FSQBUFFER2 Field - szName[1]

szName[1] (UCHAR)
Item name.

The name is an ASCIIZ string.

FSQBUFFER2 Field - szFSDName[1]

szFSDName[1] (UCHAR)
Name of the file-system driver that the item is attached to.

The name is an ASCIIZ string.

FSQBUFFER2 Field - rgFSAData[1]

rgFSAData[1] (UCHAR)
File-system driver Attach data returned by the file-system driver.

FTIME

Time data structure for file-system functions.

```
typedef struct _FTIME {
  USHORT     twosecs:5;  /*  Binary number of two-second increments. */
  USHORT     minutes:6;  /*  Binary number of minutes. */
  USHORT     hours:5;    /*  Binary number of hours. */
} FTIME;

typedef FTIME *PFTIME;
```

FTIME Field - twosecs:5

twosecs:5 (USHORT)
Binary number of two-second increments.

FTIME Field - minutes:6

minutes:6 (USHORT)
Binary number of minutes.

FTIME Field - hours:5

hours:5 (USHORT)
Binary number of hours.

GEA2

Level 3 (32-bit) (FIL_QUERYEASFROMLIST) file information - get extended attributes.

```
typedef struct _GEA2 {
  ULONG     oNextEntryOffset;  /*  Offset to next entry. */
  BYTE      cbName;            /*  Name length not including NULL. */
  CHAR      szName[1];         /*  Attribute name. */
} GEA2;

typedef GEA2 *PGEA2;
```

GEA2 Field - oNextEntryOffset

oNextEntryOffset (ULONG)
Offset to next entry.

GEA2 Field - cbName

cbName (BYTE)
Name length not including NULL.

GEA2 Field - szName[1]

szName[1] (CHAR)
Attribute name.

GEA2LIST

Get extended attributes list.

```
typedef struct _GEA2LIST {
  ULONG     cbList;   /*  Total bytes of structure including full list. */
  GEA2      list[1];  /*  Variable-length GEA2 structures. */
} GEA2LIST;

typedef GEA2LIST *PGEA2LIST;
```

GEA2LIST Field - cbList

cbList (ULONG)
Total bytes of structure including full list.

GEA2LIST Field - list[1]

list[1] (GEA2)
Variable-length GEA2 structures.

HDC

Device-context handle.

```
typedef LHANDLE HDC;
```

HDIR

Value (32-bit) used as a directory handle.

```
typedef LHANDLE HDIR;
```

HEV

Value (32-bit) used as an event semaphore handle.

```
typedef ULONG HEV;
```

HFILE

File handle.

```
typedef LHANDLE HFILE;
```

HKBD

Keyboard handle.

```
typedef unsigned short HKBD;
```

HMODULE

Module handle.

```
typedef LHANDLE HMODULE;
```

HMTX

Value (32-bit) used as a mutex semaphore handle.

```
typedef ULONG HMTX;
```

HMUX

Value (32-bit) used as a muxwait semaphore handle.

```
typedef ULONG HMUX;
```

HOTKEY

Session Manager Hot Key data structure.

```
typedef struct _HOTKEY {
  USHORT     fsHotKey;          /*  State Key Flag. */
  UCHAR      uchScancodeMake;   /*  The Scan Code of the hot key, Make. */
  UCHAR      uchScancodeBreak;  /*  The Scan Code of the hot key, Break. */
  USHORT     idHotKey;          /*  Hot Key Id. */
} HOTKEY;

typedef HOTKEY *PHOTKEY;
```

HOTKEY Field - fsHotKey

fsHotKey (USHORT)
State Key Flag.

Has the following settings:

High Byte	Bit settings are as follows:	
	Bit 15	Reserved = 0
	Bit 14	Reserved = 0
	Bit 13	Reserved = 0
	Bit 12	Reserved = 0
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down
	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
Low Byte	Bit settings are as follows:	
	Bit 7	Reserved = 0
	Bit 6	Reserved = 0
	Bit 5	Reserved = 0
	Bit 4	Reserved = 0
	Bit 3	Reserved = 0
	Bit 2	Reserved = 0
	Bit 1	Left Shift key down
	Bit 0	Right Shift key down

HOTKEY Field - uchScancodeMake

uchScancodeMake (UCHAR)
The Scan Code of the hot key, Make.

HOTKEY Field - uchScancodeBreak

uchScancodeBreak (UCHAR)
The Scan Code of the hot key, Break.

HOTKEY Field - idHotKey

idHotKey (USHORT)
Hot Key Id.

The Hot Key Id value is set by the caller. Notice that ID value FFFFh is reserved and must not be used as a Hot Key ID. See Remarks in KBD_SETSESMGRHOTKEY.

A maximum of two of the above bit selections can be selected for a given hot key definition. If more than two bits are selected or if a reserved bit is selected, the result is an INVALID_PARAMETER error code returned to the caller.

HPIPE

Value (32-bit) used as a pipe handle.

```
typedef LHANDLE HPIPE;
```

HQUEUE

Value (32-bit) used as a system queue handle.

```
typedef LHANDLE HQUEUE;
```

HRGN

Region handle.

```
typedef LHANDLE HRGN;
```

HSEM

Semaphore handle.

```
typedef VOID *HSEM;
```

HSPINLOCK

A value (32-bit) used as a handle to a spin lock.

```
#define HSPINLOCK ULONG
```

HTIMER

Value (32-bit) used as a timer handle.

```
typedef LHANDLE HTIMER;
```

HVDD

32-bit value used as a virtual device driver handle.

```
typedef LHANDLE HVDD;
```

KBDHWID

Keyboard hardware ID structure.

```
typedef struct _KBDHWID {
  USHORT     cb;         /*  Size, in bytes, of this data structure. */
  USHORT     idKbd;      /*  Keyboard hardware ID */
  USHORT     idSecond;   /*  Secondary ID */
} KBDHWID;

typedef KBDHWID *PKBDHWID;
```

KBDHWID Field - cb

cb (USHORT)
Size, in bytes, of this data structure.

On input, this field should contain the length of the Keyboard ID structure. The minimum input length value allowed is 2. On output, this field contains the actual number of bytes returned.

KBDHWID Field - idKbd

idKbd (USHORT)
Keyboard hardware ID

OS/2 supported keyboards and their hardware-generated IDs are as follows:

ID	**Keyboard**
0000H	Undetermined keyboard type
0001H	PC-AT Standard Keyboard
AB41H	101 Key Enhanced Keyboard
AB41H	102 Key Enhanced Keyboard
AB54H	88 and 89 Key Enhanced Keyboard
AB85H	122 Key Enhanced Keyboard

KBDHWID Field - idSecond

idSecond (USHORT)
Secondary ID

KBDINFO

Keyboard status data structure.

```
typedef struct _KBDINFO {
  USHORT     cb;            /*  Length, in bytes, of this data structure. */
  USHORT     fsMask;        /*  State mask. */
  USHORT     chTurnAround;  /*  Turnaround character. */
  USHORT     fsInterim;     /*  Interim character state and NLS shift state. */
  USHORT     fsState;       /*  Current shift state. */
} KBDINFO;

typedef KBDINFO *PKBDINFO;
```

KBDINFO Field - cb

cb (USHORT)
Length, in bytes, of this data structure.

10	Only valid value.

KBDINFO Field - fsMask

fsMask (USHORT)
State mask.

The system state altered by this call. If bits 0 and 1 are off, the echo state of the system is not altered. If bits 2 and 3 are off, the binary and ASCII state of the system is not altered. If bits 0 and 1 are on, or if bits 2 and 3 are on, the function returns an error. If binary mode is set, echo is ignored.

Bits 15-9	Reserved.
Bit 8	Shift return is on.
Bit 7	Length of the turn-around character (meaningful only if bit 6 is on).
Bit 6	Turn-around character is modified.
Bit 5	Interim character flags are modified.
Bit 4	Shift state is modified.
Bit 3	ASCII mode is on.
Bit 2	Binary mode is on.
Bit 1	Echo off.
Bit 0	Echo on.

KBDINFO Field - chTurnAround

chTurnAround (USHORT)
Turnaround character.

Definition of the turn-around character. In ASCII and extended-ASCII format, the turn-around character is defined as the carriage return. In ASCII format only, the turn-around character is defined in the low-order byte.

KBDINFO Field - fsInterim

fsInterim (USHORT)
Interim character state and NLS shift state.

Bits 15-8	NLS shift state. The meaning of the NLS shift varies by language. The following bits are defined to access this data: NLSS_NLS1 (0x01) - Fullwidth, National layer NLSS_NLS2 (0x02) - Katakana, JAMO phonetic NLSS_NLS3 (0x04) - Hiragana, Hangeul, TsangJye NLSS_APPL (0x10) - Application bit NLSS_NLS4 (0x40) - Romanji, HanjaCsr NLSS_KANJI (0x80) - Kanji, Hanji
Bit 7	Interim character flag is on.
Bit 6	Reserved.
Bit 5	Application requested immediate conversion.
Bits 4-0	Reserved.

KBDINFO Field - fsState

fsState (USHORT)
Current shift state.

Bit 15	SysReq key down.
Bit 14	CapsLock key down.
Bit 13	NumLock key down.
Bit 12	ScrollLock key down.
Bit 11	Right Alt key down.
Bit 10	Right Ctrl key down.
Bit 9	Left Alt key down.
Bit 8	Left Ctrl key down.

Bit 7	Insert on.
Bit 6	CapsLock on.
Bit 5	NumLock on.
Bit 4	ScrollLock on.
Bit 3	Either Alt key down.
Bit 2	Either Ctrl key down.
Bit 1	Left Shift key down.
Bit 0	Right Shift key down.

KBDKEYINFO

The Character data structure of the API function KbdCharIn.

```
typedef struct _KBDKEYINFO {
  UCHAR      chChar;     /*  ASCII Character code. */
  UCHAR      chScan;     /*  Code received for the keyboard. */
  UCHAR      fbStatus;   /*  State of the keystroke event flag. */
  UCHAR      bNlsShift;  /*  NLS shift status. Reserved, must be 0. */
  USHORT     fsState;    /*  Shift key status flag. */
  ULONG      time;       /*  Time stamp in milliseconds. */
} KBDKEYINFO;

typedef KBDKEYINFO *PKBDKEYINFO;
```

KBDKEYINFO Field - chChar

chChar (UCHAR)
ASCII Character code.

The scan code received from the keyboard is translated to the ASCII character code.

KBDKEYINFO Field - chScan

chScan (UCHAR)
Code received for the keyboard.

Scan code received from the keyboard is translated to the ASCII character code.

KBDKEYINFO Field - fbStatus

fbStatus (UCHAR)
State of the keystroke event flag.

Bits 7-6	Has the following values:

00	Undefined.
01	Final character; interim character flag is turned *off*.
10	Interim character.
11	Final character; interim character flag is turned *on*.

Bit 5	If set to 1, immediate conversion requested.
Bits 4-2	Reserved.
Bit 1	Has the following values:

0	Scan code is a character
1	Scan code is not a character; instead it is an extended key code from the keyboard.

Bit 0	If set to 1, shift status returned without a character.

KBDKEYINFO Field - bNlsShift

bNlsShift (UCHAR)
NLS shift status. Reserved, must be 0.

KBDKEYINFO Field - fsState

fsState (USHORT)
Shift key status flag.

Values are:

Bit 15	SysReq key down
Bit 14	Caps Lock key down
Bit 13	NumLock key down
Bit 12	Scroll Lock key down
Bit 11	Right Alt key down
Bit 10	Right Ctrl key down
Bit 9	Left Alt key down
Bit 8	Left Ctrl key down
Bit 7	Insert *on*
Bit 6	Caps Lock *on*
Bit 5	NumLock *on*.
Bit 4	Scroll Lock *on*
Bit 3	Either Alt key down
Bit 2	Either Ctrl key down
Bit 1	Left Shift key down
Bit 0	Right Shift key down

KBDKEYINFO Field - time

time (ULONG)

Time stamp in milliseconds.

Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.

LDTADDRINFO

LDT Address Information data structure.

```
typedef struct _LDTADDRINFO {
  PULONG     pulPhysAddr;  /*  Physical address. */
  USHORT     cb;           /*  Length. */
} LDTADDRINFO;

typedef LDTADDRINFO *PLDTADDRINFO;
```

LDTADDRINFO Field - pulPhysAddr

pulPhysAddr (PULONG)
Physical address.

LDTADDRINFO Field - cb

cb (USHORT)
Length.

LINECONTROL

Line characteristics.

```
typedef struct _LINECONTROL {
  BYTE     bDataBits;    /*  Data bits flag. */
  BYTE     bParity;      /*  Parity flag. */
  BYTE     bStopBits;    /*  Stop bit flag. */
  BYTE     fTransBreak;  /*  Transmitting break flag. */
} LINECONTROL;

typedef LINECONTROL *PLINECONTROL;
```

LINECONTROL Field - bDataBits

bDataBits (BYTE)

Data bits flag.

0x00-0x04
: Reserved

0x05
: 5 data bits

0x06
: 6 data bits

0x07
: 7 data bits (initial value)

0x08
: 8 data bits

0x09-0xFF
: Reserved

LINECONTROL Field - bParity

bParity (BYTE)
Parity flag.

0x00
: No parity

0x01
: Odd parity

0x02
: Even parity (initial value)

0x03
: Mark parity (parity bit always 1)

0x04
: Space parity (parity bit always 0)

0x05-0xFF
: Reserved

LINECONTROL Field - bStopBits

bStopBits (BYTE)
Stop bit flag.

0x00
: 1 stop bit (initial value)

0x01
: 1.5 stop bits (valid with 5-bit WORD length only)

0x02
: 2 stop bits (not valid with 5-bit WORD length)

0x03-0xFF
: Reserved

LINECONTROL Field - fTransBreak

fTransBreak (BYTE)
Transmitting break flag.

0
: Not currently transmitting break.

1

Currently transmitting break.

LONG

Signed integer in the range -2 147 483 648 through 2 147 483 647.

```
#define LONG long
```

Note: Where this data type represents a graphic coordinate in world or model space, its value is restricted to -134 217 728 through 134 217 727.

A graphic coordinate in device or screen coordinates is restricted to -32 768 through 32 767.

The value of a graphic coordinate may be further restricted by any transforms currently in force, including the positioning of the origin of the window on the screen. In particular, coordinates in world or model space must not generate coordinate values after transformation (that is, in device or screen space) outside the range -32 768 through 32 767.

MODEMSTATUS

Modem Control data structure.

```
typedef struct _MODEMSTATUS {
  BYTE     fbModemOn;   /*  Modem Control Signals ON Mask. */
  BYTE     fbModemOff;  /*  Modem Control Signals OFF Mask. */
} MODEMSTATUS;

typedef MODEMSTATUS *PMODEMSTATUS;
```

MODEMSTATUS Field - fbModemOn

fbModemOn (BYTE)
Modem Control Signals ON Mask.

The physical device driver sets the modem control signals as defined in this field. Bit 0 is DTR; bit 1 is RTS. If any other bits are set/reset by the masks, then a general failure error results. The OFF mask contains a mask (with all bits equal to 0) to turn the modem control signal *off*. The ON mask contains a mask (with all bits equal to 1) to turn the modem control signal *on*. If the Parameter Packet indicates both turn *off* and turn *on* on the same bit, the bit will be turned *on*.

For example:

```
 Mask ON   Mask OFF  Description

    01h       FFh    Set DTR

    00h       FEh    Clear DTR

    02h       FFh    Set RTS

    00h       FDh    Clear RTS

    03h       FFh    Set DTR and RTS

    00h       FCh    Clear DTR and
```

If DTR Control mode input handshaking, RTS Control mode input handshaking or *toggling on transmit* is set, then this request cannot try to change the state of the modem control signals, that are being used for input handshaking or toggling on transmit. If the request tries to modify a modem control signal that is being used for input handshaking or toggling on transmit, a *general failure* error results.

MODEMSTATUS Field - fbModemOff

fbModemOff (BYTE)
Modem Control Signals OFF Mask.

See Modem Control Signals ON Mask above.

MONITORPOSITION

Monitor Position data structure.

```
typedef struct _MONITORPOSITION {
  USHORT     fPosition;  /*  Placement flag. */
  USHORT     index;      /*  Index. */
  ULONG      pbInBuf;    /*  Address of the Input Buffer. */
  USHORT     offOutBuf;  /*  Offset of the Output Buffer. */
} MONITORPOSITION;

typedef MONITORPOSITION *PMONITORPOSITION;
```

MONITORPOSITION Field - fPosition

fPosition (USHORT)
Placement flag.

The DosMonReg function call parameter that is used by an application to indicate:

- Where its monitor buffers are to be placed within a monitor chain relative to monitors already registered on the monitor chain
- What special processing requirements need to be supported by the monitor dispatcher.

Refer to the DosMonReg function description in the *OS/2 Control Program Programming Reference* for valid parameter values.

MONITORPOSITION Field - index

index (USHORT)
Index.

Used by an application to indicate on which monitor chain its monitor buffers are being registered. The accepted values for this parameter vary for each device driver. See information on the physical mouse device driver, the physical keyboard device driver and

the physical parallel port device driver in the *OS/2 Input/Output Device Driver Reference* to determine the valid parameter value for each device driver.

Refer to the descriptions of the DosMonReg, DosMonRead, and DosMonWrite functions in the *OS/2 Control Program Programming Reference* for more information.

MONITORPOSITION Field - pbInBuf

pbInBuf (ULONG)
Address of the Input Buffer.

Specifies the address of a monitor input buffer allocated by an application, initialized by the monitor dispatcher, and used by the DosMonRead function.

MONITORPOSITION Field - offOutBuf

offOutBuf (USHORT)
Offset of the Output Buffer.

Specifies the offset of a monitor output buffer allocated by an application in the same data segment as the input buffer, initialized by the monitor dispatcher, and used by the DosMonWrite function.

MOUEVENTINFO

Mouse event queue data structure.

```
typedef struct _MOUEVENTINFO {
  ULONG     fs;    /*  Mouse state. */
  LONG      row;   /*  Horizontal position. */
  LONG      col;   /*  Vertical position. */
  ULONG     time;  /*  Timestamp. */
} MOUEVENTINFO;

typedef MOUEVENTINFO *PMOUEVENTINFO;
```

MOUEVENTINFO Field - fs

fs (ULONG)
Mouse state.

The state of the mouse at the time of the event.

Bit	Description
31-7	Reserved; set to zero.
6	Set if button 3 is down.
5	Set if mouse is moving and button 3 is down.
4	Set if button 2 is down.

3	Set if mouse is moving and button 2 is down.
2	Set if button 1 is down.
1	Set if mouse is moving and button 1 is down.
0	Set if mouse is moving and no buttons are down.

MOUEVENTINFO Field - row

row (LONG)
Horizontal position.

The absolute or relative row position.

MOUEVENTINFO Field - col

col (LONG)
Vertical position.

The absolute or relative column position.

MOUEVENTINFO Field - time

time (ULONG)
Timestamp.

Time stamp.

MOUQUEINFO

Mouse queue status structure.

```
typedef struct _MOUQUEINFO {
  ULONG     cEvents;     /*  Current number of queue elements. */
  ULONG     cmaxEvents;  /*  Maximum number of queue elements. */
} MOUQUEINFO;

typedef MOUQUEINFO *PMOUQUEINFO;
```

MOUQUEINFO Field - cEvents

cEvents (ULONG)
Current number of queue elements.

MOUQUEINFO Field - cmaxEvents

cmaxEvents (ULONG)
Maximum number of queue elements.

NOPTRRECT

Pointer shape collision area data structure.

```
typedef struct _NOPTRRECT {
  ULONG     row;   /*  Upper-left row coordinate. */
  ULONG     col;   /*  Upper-left column coordinate. */
  ULONG     cRow;  /*  Lower-right row coordinate. */
  ULONG     cCol;  /*  Lower-right column coordinate. */
} NOPTRRECT;

typedef NOPTRRECT *PNOPTRRECT;
```

NOPTRRECT Field - row

row (ULONG)
Upper-left row coordinate.

NOPTRRECT Field - col

col (ULONG)
Upper-left column coordinate.

NOPTRRECT Field - cRow

cRow (ULONG)
Lower-right row coordinate.

NOPTRRECT Field - cCol

cCol (ULONG)
Lower-right column coordinate.

NPCH

Pointer (32-bit) to a value or array of values.

```
typedef unsigned char *NPCH;
```

NPCHAR

Near pointer to a character.

```
typedef CHAR NEAR *NPCHAR;
```

NPFN

Pointer to a procedure.

```
typedef _NPFN *NPFN;
```

In the header file, this is a two-part definition as shown below:

```
typedef  int    (APIENTRY _NPFN) ();
typedef  _NPFN  *NPFN;
```

NPSZ

Pointer (32-bit) to a null-terminated string.

```
typedef unsigned char *NPSZ;
```

OEMINFO

OEM information data structure.

```
typedef struct _OEMINFO {
  ULONG      OEMLength;         /*  Length */
  USHORT     Manufacturer;      /*  Manufacturer Specific Data */
  ULONG      ManufacturerData;  /*  Manufacturer specific data */
} OEMINFO;
```

OEMINFO Field - OEMLength

OEMLength (ULONG)
Length

Represents the combined length of all parameter packet fields. This is a required field for all calls, including those made by way of the 32-bit DosDevIOCtl, with a minimum packet length of 10 bytes.

OEMINFO Field - Manufacturer

Manufacturer (USHORT)
Manufacturer Specific Data

Identifies the name of the manufacturer.

OEMINFO Field - ManufacturerData

ManufacturerData (ULONG)
Manufacturer specific data

Manufacturer-specific data is in a format designated by the manufacturer.

OEMSVGAINFO

OEM graphic adapter information data structure.

```
typedef struct _OEMSVGAINFO {
  USHORT     AdapterType;  /*  Adapter type. */
  USHORT     ChipType;     /*  Chip Type. */
  ULONG      Memory;       /*  Video memory. */
} OEMSVGAINFO;
```

OEMSVGAINFO Field - AdapterType

AdapterType (USHORT)
Adapter type.

Value of the chip set manufacturer.

OEMSVGAINFO Field - ChipType

ChipType (USHORT)
Chip Type.

Chip type value according to the specific manufacturer.

OEMSVGAINFO Field - Memory

Memory (ULONG)
Video memory.

Memory size of the video adapter detected.

PCH

Pointer to a character string.

```
typedef unsigned char *PCH;
```

PCSZ

Pointer to a constant null-terminated string.

```
typedef const char *PCSZ;
```

PFHLOCK

Pointer to an unsigned integer in the range 0 through 4 294 967 295.

```
typedef PULONG PFHLOCK;
```

PFN

Pointer to a procedure.

```
typedef _PFN *PFN;
```

In the header file, this is a two-part definition as shown below:

```
typedef  int   (APIENTRY _PFN) ();
typedef  _PFN  *PFN;
```

PFNEXITLIST

Address of a routine to be executed.

```
typedef FNEXITLIST *PFNEXITLIST;
```

In the header files, the FNEXITLIST structure is defined as shown below:

```
typedef VOID APIENTRY FNEXITLIST (ULONG);
```

PFNSIGHANDLER

Pointer (32-bit) to a function with *pascal* calling type.

```
typedef CHAR *PFNSIGHANDLER;
```

PFNTHREAD

Address of the code to be executed when the thread begins execution.

```
typedef FNTHREAD *PFNTHREAD;
```

In the header files, the FNTHREAD structure is defined as shown below:

```
typedef VOID APIENTRY FNTHREAD (ULONG);
```

PIB

Process Information Block structure.

```
typedef struct _PIB {
  ULONG     pib_ulpid;     /*  Process identifier. */
  ULONG     pib_ulppid;    /*  Parent process identifier. */
  ULONG     pib_hmte;      /*  Module handle of executable program. */
  PCHAR     pib_pchcmd;    /*  Command line pointer. */
  PCHAR     pib_pchenv;    /*  Environment pointer. */
  ULONG     pib_flstatus;  /*  Process' status bits. */
  ULONG     pib_ultype;    /*  Process' type code. */
} PIB;

typedef PIB *PPIB;
```

An OS/2 application that has been loaded into memory and prepared for execution is called a *process* . A process is the code, data, and other resources of an application, such as file handles, semaphores, pipes, queues, and so on. The OS/2 operating system considers every application it loads to be a process.

Information about a process is kept in a read/write area of the process address space, called the Process Information Block (PIB). The operating system creates and maintains a PIB for every process in the system.

An application can access the PIB of a specific process using DosGetInfoBlocks.

PIB Field - pib_ulpid

pib_ulpid (ULONG)
Process identifier.

PIB Field - pib_ulppid

pib_ulppid (ULONG)
Parent process identifier.

PIB Field - pib_hmte

pib_hmte (ULONG)
Module handle of executable program.

PIB Field - pib_pchcmd

pib_pchcmd (PCHAR)
Command line pointer.

This is the address of the ASCIIZ argument strings passed to the program. This string represents command parameters.

The convention used by CMD.EXE is that the first of these strings is the program name (as entered from the command prompt or found

in a batch file), and the second string consists of the parameters for the program. The second ASCIIZ string is followed by an additional byte of zeros. A value of zero for the address of *pib_pchcmd* means that no arguments are to be passed to the program.

PIB Field - pib_pchenv

pib_pchenv (PCHAR)
Environment pointer.

These strings represent environment variables and their current values. An environment string has the following form:

```
variable=value
```

The last ASCIIZ environment string must be followed by an additional byte of zeros.

PIB Field - pib_flstatus

pib_flstatus (ULONG)
Process' status bits.

A value of 1 in this bit flag indicates that the current process is in exit list processing.

PIB Field - pib_ultype

pib_ultype (ULONG)
Process' type code.

The following process' type codes are available:

0	Full screen protect-mode session
1	Requires real mode. Dos emulation.
2	VIO windowable protect-mode session
3	Presentation Manager protect-mode session
4	Detached protect-mode process.

PID

Process identity.

```
typedef LHANDLE PID;
```

PIPEINFO

DosQueryNPipeInfo uses this data structure for level-1 information.

```
typedef struct _PIPEINFO {
  USHORT     cbOut;       /*  Actual size of the buffer for outbound data. */
  USHORT     cbIn;        /*  Actual size of the buffer for inbound data. */
  BYTE       cbMaxInst;   /*  Maximum number of pipe instances. */
  BYTE       cbCurInst;   /*  Current number of pipe instances. */
  BYTE       cbName;      /*  Length of szName. */
  CHAR       szName[1];   /*  Name of the pipe. */
} PIPEINFO;

typedef PIPEINFO *PIPEINFO;
```

PIPEINFO Field - cbOut

cbOut (USHORT)
Actual size of the buffer for outbound data.

PIPEINFO Field - cbIn

cbIn (USHORT)
Actual size of the buffer for inbound data.

PIPEINFO Field - cbMaxInst

cbMaxInst (BYTE)
Maximum number of pipe instances.

PIPEINFO Field - cbCurInst

cbCurInst (BYTE)
Current number of pipe instances.

PIPEINFO Field - cbName

cbName (BYTE)
Length of *szName* .

PIPEINFO Field - szName[1]

szName[1] (CHAR)
Name of the pipe.

The name of the pipe (including \\ComputerName if the pipe is on a remote system).

PIPESEMSTATE

Data structure for the status of a named pipe that is attached to a semaphore.

```
typedef struct _PIPESEMSTATE {
  BYTE       fStatus;  /*  A coded value that indicates the status of the named pipe. */
  BYTE       fFlag;    /*  Additional information about the state of the named pipe. */
  USHORT     usKey;    /*  A key value that distinguishes events arriving on different named pipes that are atta
  USHORT     usAvail;  /*  Number of bytes available in the pipe. */
} PIPESEMSTATE;

typedef PIPESEMSTATE *PPIPESEMSTATE;
```

PIPESEMSTATE Field - fStatus

fStatus (BYTE)
A coded value that indicates the status of the named pipe.

The following codes are available:

0	NPSS_EOI End of information buffer. No more information records follow, and subsequent fields in this information record have no defined value.
1	NPSS_RDATA Read data is available.
2	NPSS_WSPACE Write space is available.
3	NPSS_CLOSE The pipe is closed.

PIPESEMSTATE Field - fFlag

fFlag (BYTE)
Additional information about the state of the named pipe.

This parameter contains the following bit fields:

Bit	Description

7-1	Reserved
0	NPSS_WAIT If set, a thread is waiting at the other end of the pipe.

PIPESEMSTATE Field - usKey

usKey (USHORT)
A key value that distinguishes events arriving on different named pipes that are attached to the same semaphore.

This value is specified in the *key* field when DosSetNPipeSem is issued.

PIPESEMSTATE Field - usAvail

usAvail (USHORT)
Number of bytes available in the pipe.

If *fStatus* has a value of 1, this field contains the number of bytes of data that are available to read from the pipe. If *fStatus* has a value of 2, this field contains the number of bytes of write space that are available in the pipe.

PSZ

Pointer to a null-terminated string.

If you are using C++ **, you may need to use PCSZ.

```
typedef unsigned char *PSZ;
```

PTRDRAWDATA

Pointer Draw Data structure.

```
typedef struct _PTRDRAWDATA {
  USHORT     cb;         /*  Length of the data structure. */
  USHORT     usConfig;   /*  Display Configuration Number. */
  USHORT     usFlag;     /*  Caller flag. */
} PTRDRAWDATA;

typedef PTRDRAWDATA *PPTRDRAWDATA;
```

PTRDRAWDATA Field - cb

cb (USHORT)
Length of the data structure.

The length of the data structure is equal to 6.

PTRDRAWDATA Field - usConfig

usConfig (USHORT)
Display Configuration Number.

The display configuration number on which the pointer draw routine should draw.

PTRDRAWDATA Field - usFlag

usFlag (USHORT)
Caller flag.

Specifies whether this call is made for an application or the Base Video Subsystem (BVS), where:

0	Application
1	BVS

PTRDRAWFUNCTION

Pointer Draw Routine access data structure.

```
typedef struct _PTRDRAWFUNCTION {
  USHORT     usReturnCode;  /*  Return Code. */
  PFN        pfnDraw;       /*  Pointer to Draw Routine Entry Point (Selector:offset). */
  PCH        pchDataSeg;    /*  Pointer to Draw Routine Data Segment Selector. */
} PTRDRAWFUNCTION;

typedef PTRDRAWFUNCTION *PPTRDRAWFUNCTION;
```

PTRDRAWFUNCTION Field - usReturnCode

usReturnCode (USHORT)
Return Code.

PTRDRAWFUNCTION Field - pfnDraw

pfnDraw (PFN)
Pointer to Draw Routine Entry Point (Selector:offset).

PTRDRAWFUNCTION Field - pchDataSeg

pchDataSeg (PCH)
Pointer to Draw Routine Data Segment Selector.

PTRLOC

Mouse pointer position.

```
typedef struct _PTRLOC {
  ULONG     row;  /*  Pointer to the row screen position. */
  ULONG     col;  /*  Pointer to the column screen position. */
} PTRLOC;

typedef PTRLOC *PPTRLOC;
```

PTRLOC Field - row

row (ULONG)
Pointer to the row screen position.

The current pointer row coordinate (pels or characters).

PTRLOC Field - col

col (ULONG)
Pointer to the column screen position.

The current pointer column coordinate (pels or characters).

PTRSHAPE

The address of a structure, in application storage, where the application stores the data necessary for the pointer device driver to return information about the Row-by-Column image for each bit plane for the mode the display is currently running. See MouSetPtrShape for a further description of the contents of this structure.

```
typedef struct _PTRSHAPE {
  USHORT     cb;       /*  Length of the pointer buffer. */
  USHORT     col;      /*  The number of columns in mouse shape. */
  USHORT     row;      /*  The number of rows in mouse shape. */
  USHORT     colHot;   /*  Pointer image hotspot. */
  USHORT     rowHot;   /*  Pointer image hotspot. */
} PTRSHAPE;

typedef PTRSHAPE *PPTRSHAPE;
```

PTRSHAPE Field - cb

cb (USHORT)
Length of the pointer buffer.

The length of the pointer buffer available for the pointer device driver to build a row-by-column image for each bit plane for the mode the display is currently running. This value is supplied by the application. If the value is too small, pointer draw places the true length of the image in this field and returns an error.

For all OS/2 system-supported modes, *cb* is specified in bytes and is equal to:

Mono and Text Modes — For text mode, height and width must be 1, so length is always 4.

```
cb = (height in chars) * (width in chars) * 2 * 2
   = 1 * 1 * 2 * 2
   = 4
```

Graphics Mode — Width-in-pels must be a multiple of 8.

```
cb = (height in pels) * (width in pels) * (bits per pel)
```

PTRSHAPE Field - col

col (USHORT)
The number of columns in mouse shape.

In graphics modes, this field contains the pel width (columns) of the mouse shape for the session and must be greater than or equal to 1. In text modes, col must equal 1.

PTRSHAPE Field - row

row (USHORT)
The number of rows in mouse shape.

In graphics modes, this field contains the pel height (rows) of the mouse shape for the session and must be greater than or equal to 1. In text modes, row must equal 1.

PTRSHAPE Field - colHot

colHot (USHORT)
Pointer image hotspot.

This value is returned by the mouse device driver to indicate the relative column offset within the pointer image. The value defines the center (hotspot) of the pointer image. This value is a signed number that represents either character or pel offset, depending on the display mode.

PTRSHAPE Field - rowHot

rowHot (USHORT)
Pointer image hotspot.

The row of the pointer image hotspot. This value is returned by the mouse device driver to indicate the relative row offset within the pointer image. The value defines the center (hotspot) of the pointer image. This value is a signed number that represents either character or pel offset, depending on the display mode.

PVOID

Pointer to a data type of undefined format.

```
typedef VOID *PVOID;
```

QWORD

Quad word structure.

```
typedef struct _QWORD {
  ULONG     ulLo;  /*  Low word. */
  ULONG     ulHi;  /*  High word. */
} QWORD;

typedef QWORD *PQWORD;
```

QWORD Field - ulLo

ulLo (ULONG)
Low word.

QWORD Field - ulHi

ulHi (ULONG)
High word.

RATEDELAY

Typematic Rate and Delay data structure.

```
typedef struct _RATEDELAY {
  USHORT     usDelay;  /*  Typematic delay. */
  USHORT     usRate;   /*  Typematic rate. */
} RATEDELAY;

typedef RATEDELAY *PRATEDELAY;
```

RATEDELAY Field - usDelay

usDelay (USHORT)
Typematic delay.

Specifies the typematic delay in milliseconds. A value greater than the maximum value defaults to the maximum value.

RATEDELAY Field - usRate

usRate (USHORT)
Typematic rate.

Specifies the typematic rate in characters per second. A value greater than the maximum value defaults to the maximum value.

REQUESTDATA

REQUESTDATA data structure.

```
typedef struct _REQUESTDATA {
  PID       pid;     /*  Process identifier of the process that placed the element into the queue. */
  ULONG     ulData;  /*  An event code that is specified by the application. */
} REQUESTDATA;

typedef REQUESTDATA *PREQUESTDATA;
```

REQUESTDATA Field - pid

pid (PID)
Process identifier of the process that placed the element into the queue.

REQUESTDATA Field - ulData

ulData (ULONG)
An event code that is specified by the application.

This data is understood by both the thread that is adding the element to the queue (the client thread) and the thread that receives the queue element (server thread). It has no special meaning, and is not altered by the operating system.

RESULTCODES

RESULTCODES data structure.

```
typedef struct _RESULTCODES {
  ULONG     codeTerminate;  /*  Termination code or process identifier. */
  ULONG     codeResult;     /*  Exit code. */
} RESULTCODES;

typedef RESULTCODES *PRESULTCODES;
```

RESULTCODES Field - codeTerminate

codeTerminate (ULONG)
Termination code or process identifier.

For asynchronous requests, this field contains the process identifier of the child process. For synchronous requests, this field contains the termination code furnished by the system describing why the child process ended. The values of the termination code are shown in the following list:

0	TC_EXIT Normal exit.
1	TC_HARDERROR Hard-error halt
2	TC_TRAP Trap operation for a 16-bit child process
3	TC_KILLPROCESS Unintercepted DosKillProcess
4	TC_EXCEPTION Exception operation for a 32-bit child process

RESULTCODES Field - codeResult

codeResult (ULONG)
Exit code.

Result code specified by the terminating synchronous process on its last call to DosExit.

RXQUEUE

Receive/Transmit Queue Information data structure.

```
typedef struct _RXQUEUE {
  USHORT     cch;  /*  Number of characters in the queue. */
  USHORT     cb;   /*  Size of receive/transmit queue. */
} RXQUEUE;

typedef RXQUEUE *PRXQUEUE;
```

RXQUEUE Field - cch

cch (USHORT)
Number of characters in the queue.

Binary integer with the number of characters received/to be transmitted in the physical device driver receive/transmit queue. This is a memory buffer between the memory pointed to by the READ/WRITE request packet and the receive/transmit hardware for this COM device.

Note: The behavior of data movement between the READ/WRITE request and the receive/transmit queue can change with each release of the physical device driver. Applications should not be dependent on this information.

RXQUEUE Field - cb

cb (USHORT)
Size of receive/transmit queue.

Binary integer with the size of the physical device driver receive/transmit queue. Applications should be independent of the receive/transmit queue size (fixed or not fixed). The information in this field allows the application to get the size of the receive/transmit queue, but is subject to change. The current size of the receive queue is approximately 1KB.

Using this information, the application should avoid device driver receive queue overruns by using an application-to-application block protocol with the system communicating with the application.

SCALEFACT

The address of the control block structure that contains the current row and column coordinate scaling factors. The scaling factors must be greater than or equal to 1 and less than or equal to (32K - 1).

```
typedef struct _SCALEFACT {
```

```
  ULONG     rowScale;  /*  Row scaling factor. */
  ULONG     colScale;  /*  Column coordinate scaling factor. */
} SCALEFACT;

typedef SCALEFACT *PSCALEFACT;
```

SCALEFACT Field - rowScale

rowScale (ULONG)
Row scaling factor.

SCALEFACT Field - colScale

colScale (ULONG)
Column coordinate scaling factor.

SEL

Selector.

```
typedef USHORT SEL;
```

SEMRECORD

Muxwait semaphore data structure.

```
typedef struct _SEMRECORD {
  HSEM      hsemCur;  /*  Handle of the semaphore. */
  ULONG     ulUser;   /*  User-defined value. */
} SEMRECORD;

typedef SEMRECORD *PSEMRECORD;
```

SEMRECORD Field - hsemCur

hsemCur (HSEM)
Handle of the semaphore.

SEMRECORD Field - ulUser

ulUser (ULONG)
User-defined value.

SGID

This data type is specific to versions, higher than version 3, of the OS/2 operating system.

Value used to hold a screen group ID.

```
#ifdef _PPC_
typedef ULONG SGID;
#else
typedef USHORT SGID;
#endif
```

SHIFTSTATE

Shift State data structure.

```
typedef struct _SHIFTSTATE {
  USHORT     fsState;  /*  Shift state flag. */
  BYTE       fNLS;     /*  NLS shift status flag. */
} SHIFTSTATE;

typedef SHIFTSTATE *PSHIFTSTATE;
```

SHIFTSTATE Field - fsState

fsState (USHORT)
Shift state flag.

High Byte	Has the following settings:	
	Bit 15	SysReq key down
	Bit 14	Caps Lock key down
	Bit 13	NumLock key down
	Bit 12	ScrollLock key down
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down
	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
Low Byte	Has the following:	
	Bit 7	Insert *on*
	Bit 6	Caps Lock *on*

Bit 5	NumLock *on*
Bit 4	ScrollLock *on*
Bit 3	Either Alt key down
Bit 2	Either Ctrl key down
Bit 1	Left Shift key down
Bit 0	Right Shift key down

SHIFTSTATE Field - fNLS

fNLS (BYTE)
NLS shift status flag.

SHORT

Signed integer in the range -32 768 through 32 767.

```
#define SHORT short
```

STARTDATA

Start session data structure.

```
typedef struct _STARTDATA {
  USHORT     Length;         /*  The length of the data structure, in bytes, including Length itself. */
  USHORT     Related;        /*  An indicator which specifies whether the session created is related to the call
  USHORT     FgBg;           /*  An indicator which specifies whether the new session should be started in the f
  USHORT     TraceOpt;       /*  An indicator which specifies whether the program started in the new session sho
for tracing. */
  PSZ        PgmTitle;       /*  Address of an ASCIIZ string that contains the program title. */
  PSZ        PgmName;        /*  The address of an ASCIIZ string that contains the file specification of the pro
  PBYTE      PgmInputs;      /*  Either 0 or the address of an ASCIIZ string that contains the input arguments t
  PBYTE      TermQ;          /*  Either 0 or the address of an ASCIIZ string that contains the file specificatio
  PBYTE      Environment;    /*  The address of an environment string to be passed to the program started in the
  USHORT     InheritOpt;     /*  Specifies whether the program started in the new session should inherit the cal
nd open file handles. */
  USHORT     SessionType;    /*  The type of session that should be created for this program. */
  PSZ        IconFile;       /*  Either 0 or the address of an ASCIIZ string that contains the file specificatio
  ULONG      PgmHandle;      /*  Either 0 or the program handle. */
  USHORT     PgmControl;     /*  An indicator which specifies the initial state for a windowed application. */
  USHORT     InitXPos;       /*  The initial x-coordinate, in pels, for the initial session window. */
  USHORT     InitYPos;       /*  The initial y-coordinate, in pels, for the initial session window. */
  USHORT     InitXSize;      /*  The initial x extent, in pels, for the initial session window. */
  USHORT     InitYSize;      /*  The initial y extent, in pels, for the initial session window. */
  USHORT     Reserved;       /*  Reserved; must be zero. */
  PSZ        ObjectBuffer;   /*  Buffer in which the name of the object that contributed to the failure of DosEx
 is returned. */
  ULONG      ObjectBuffLen;  /*  The length, in bytes, of the buffer pointed to by ObjectBuffer. */
} STARTDATA;

typedef STARTDATA *PSTARTDATA;
```

STARTDATA Field - Length

Length (USHORT)
The length of the data structure, in bytes, including *Length* itself.

A length of at least 32 bytes must be used to start a DOS session with the session type specified. A length greater than 32 is not allowed if the Session Manager detects that the Presentation Manager is not present.

When a *Length* of 24 or 30 bytes is specified, DosStartSession initializes the missing parameters to 0. This allows the Shell to provide values for the missing information, based on the installation file entry for the program being started.

Specify a *Length* of 30 bytes to use the environment and inheritance features of the system. Specify a *Length* of 50 bytes to specify the type of session to start, and to define data for windows.

A *Length* of 60 bytes allows you to use all of the functions provided by DosStartSession.

STARTDATA Field - Related

Related (USHORT)
An indicator which specifies whether the session created is related to the calling session.

The values of this field are as follows:

0	SSF_RELATED_INDEPENDENT New session is an independent session (not related).
1	SSF_RELATED_CHILD New session is a child session (related).

An independent session cannot be controlled by the calling program. It may not be specified as the target of DosSelectSession, DosSetSession, or DosStopSession. The *TermQ* field is ignored for independent sessions.

Refer to **"Parent/Child Relationship"** in the **Remarks** section in DosStartSession for additional information about related sessions.

STARTDATA Field - FgBg

FgBg (USHORT)
An indicator which specifies whether the new session should be started in the foreground or background.

If a windowed session is started in the foreground, the new session will be given the window focus. The values of this field are described in the following list:

0	SSF_FGBG_FORE Start session in foreground
1	SSF_FGBG_BACK Start session in background

STARTDATA Field - TraceOpt

TraceOpt (USHORT)
An indicator which specifies whether the program started in the new session should be executed under conditions for tracing.

The values of this field are described in the following list:

0 SSF_TRACEOPT_NONE
No trace.

1 SSF_TRACEOPT_TRACE
Trace with no notification of descendants.

2 SSF_TRACEOPT_TRACEALL
Trace all descendant sessions.

Related equals 1 and a termination queue must be supplied when this value is specified.

Refer to **"Debugger Considerations"** in the **Remarks** section in DosStartSession for additional information about a tracing descendant sessions.

STARTDATA Field - PgmTitle

PgmTitle (PSZ)
Address of an ASCIIZ string that contains the program title.

The string can be up to 61 bytes long, including the terminating byte of 0. If the address specified is 0, or if the ASCIIZ string is null, then the initial title is *PgmName* minus any leading drive and path information.

STARTDATA Field - PgmName

PgmName (PSZ)
The address of an ASCIIZ string that contains the file specification of the program to be loaded.

If the address specified is 0, or if the ASCIIZ string is null, then the program identified by the *PgmHandle* is started in the new session.

Refer to **"PgmName and PgmInputs Considerations"** in the **Remarks** section in DosStartSession for additional information about a 0 *PgmName* address.

STARTDATA Field - PgmInputs

PgmInputs (PBYTE)
Either 0 or the address of an ASCIIZ string that contains the input arguments to be passed to the program.

STARTDATA Field - TermQ

TermQ (PBYTE)
Either 0 or the address of an ASCIIZ string that contains the file specification of a system queue.

This field is ignored when *Related* is set to SSF_RELATED_INDEPENDENT.

Refer to **"Parent/Child Termination Considerations"** in the **Remarks** section in DosStartSession for additional information about the *TermQ* field.

For more information about system queues, refer to DosCreateQueue.

STARTDATA Field - Environment

Environment (PBYTE)
The address of an environment string to be passed to the program started in the new session.

This parameter may be used for independent or related DosStartSession functions.

When the *Environment* field is 0, the program in the new session inherits the environment of the Shell if the *InheritOpt* field is equal to SSF_INHERTOPT_SHELL, or the environment of the program issuing DosStartSession if the *InheritOpt* field is equal to SSF_INHERTOPT_PARENT.

DOS Sessions

By default, the DOS session processes the AUTOEXEC.BAT file on the DOS startup drive. Environment variables may be set in the AUTOEXEC.BAT file or by specifying them via this parameter. Any setting available from the DOS setting notebook may also be set here. For instance, to have D:\MYEXEC.BAT run instead of AUTOEXEC.BAT, enable direct access to the hardware timer, disable the break key, and set the real memory size to 256K, specify the environment string as:

```
myenv = "DOS_AUTOEXEC=D:\\MYEXEC.BAT\0HW_TIMER=1\0DOS_BREAK=0\0DOS_RMSIZE=256\0";
```

See the *pEnv* parameter in DosExecPgm for additional information about environment settings.

STARTDATA Field - InheritOpt

InheritOpt (USHORT)
Specifies whether the program started in the new session should inherit the calling program's environment and open file handles.

The values of this field are described in the following list:

0	SSF_INHERTOPT_SHELL Inherit the Shell's environment.
1	SSF_INHERTOPT_PARENT Inherit the environment of the program issuing the DosStartSession call.

The *InheritOpt* field may be used for independent or related DosStartSession functions. Therefore, a DosStartSession function with the *InheritOpt* field equal to SSF_INHERTOPT_PARENT is equivalent to DosExecPgm, except that the new program does not inherit the priority of the parent process, or the keyboard and video characteristics associated with the parent session. Also, a parent process/child process relationship is not established.

Refer to **"Parent/Child Relationship"** in the **Remarks** section in DosStartSession for additional information about related sessions.

The *InheritOpt* field for a DOS session is different than the *InheritOpt* field for a non-DOS session. An *InheritOpt* value of SSF_INHERTOPT_PARENT for a DOS session only inherits the parent's current drive and path. It does not inherit the parent's environment.

STARTDATA Field - SessionType

SessionType (USHORT)
The type of session that should be created for this program.

The values of this field are shown in the list below:

0 SSF_TYPE_DEFAULT
Use the *PgmHandle* data, or allow the Shell to establish the session type.

1 SSF_TYPE_FULLSCREEN
Start the program in a full-screen session.

2 SSF_TYPE_WINDOWABLEVIO
Start the program in a windowed session for programs using the Base Video Subsystem.

3 SSF_TYPE_PM.
Start the program in a windowed session for programs using the Presentation Manager services (including AVIO calls).

4 SSF_TYPE_VDM
Start the program in a full-screen DOS session.

7 SSF_TYPE_WINDOWEDVDM
Start the program in a windowed DOS session.

STARTDATA Field - IconFile

IconFile (PSZ)
Either 0 or the address of an ASCIIZ string that contains the file specification of an icon definition.

If you do not provide an icon file name with DosStartSession, the system looks for an associated icon file (with a file extension of .ICO, or an extended attribute of .ICON). The system provides a default icon if an icon file name is not provided with DosStartSession.

STARTDATA Field - PgmHandle

PgmHandle (ULONG)
Either 0 or the program handle.

The program handle identifies the program in the installation file to be started, the program title, the session type, and the initial window size and position. However, information may be specified with DosStartSession to override the information in the installation file for this invocation of the program.

DosStartSession does not support program groups.

Refer to **"Program Handle Considerations"** in the **Remarks** secion of DosStartSession for additional information about program handles.

STARTDATA Field - PgmControl

PgmControl (USHORT)
An indicator which specifies the initial state for a windowed application.

This field is ignored for full-screen sessions.

The bits in this field have the following values:

Bit	Description
15	SSF_CONTROL_SETPOS (0x8000)

	Use specified size and position
4-14	Reserved
3	SSF_CONTROL_NOAUTOCLOSE (0x0008) No Auto Close
2	SSF_CONTROL_MINIMIZE (0x0004) Minimize
1	SSF_CONTROL_MAXIMIZE (0x0002) Maximize
0	SSF_CONTROL_INVISIBLE (0x0001) Invisible
0	SSF_CONTROL_VISIBLE (0x0000) Visible

Note: The "No Auto Close" bit is used only for VIO Windowable applications, and is ignored for all other types of applications.

STARTDATA Field - InitXPos

InitXPos (USHORT)
The initial x-coordinate, in pels, for the initial session window.

Coordinates (0,0) indicate the bottom left corner of the display. This field is ignored for full-screen sessions.

STARTDATA Field - InitYPos

InitYPos (USHORT)
The initial y-coordinate, in pels, for the initial session window.

Coordinates (0,0) indicate the bottom left corner of the display. This field is ignored for full-screen sessions.

STARTDATA Field - InitXSize

InitXSize (USHORT)
The initial x extent, in pels, for the initial session window.

This field is ignored for full-screen sessions.

STARTDATA Field - InitYSize

InitYSize (USHORT)
The initial y extent, in pels, for the initial session window.

This field is ignored for full-screen sessions.

STARTDATA Field - Reserved

Reserved (USHORT)
Reserved; must be zero.

STARTDATA Field - ObjectBuffer

ObjectBuffer (PSZ)
Buffer in which the name of the object that contributed to the failure of DosExecPgm is returned.

The name of the object is usually the name of a dynamic link library that could not be loaded.

DosStartSession calls DosExecPgm to start all full-screen, VIO windowed, and Presentation Manager programs.

STARTDATA Field - ObjectBuffLen

ObjectBuffLen (ULONG)
The length, in bytes, of the buffer pointed to by *ObjectBuffer* .

STATUSDATA

Status data structure.

```
typedef struct _STATUSDATA {
  USHORT     Length;     /*  The length of the data structure, in bytes, including Length itself. */
  USHORT     SelectInd;  /*  An indicator that specifies whether the target session should be flagged as selecta
  USHORT     BondInd;    /*  An indicator that specifies which session to bring to the foreground the next time
*/
} STATUSDATA;

typedef STATUSDATA *PSTATUSDATA;
```

STATUSDATA Field - Length

Length (USHORT)
The length of the data structure, in bytes, including *Length* itself.

STATUSDATA Field - SelectInd

SelectInd (USHORT)
An indicator that specifies whether the target session should be flagged as selectable or non-selectable.

Possible values are shown in the following list:

0 SET_SESSION_UNCHANGED
Leaves the current setting unchanged.

1 SET_SESSION_SELECTABLE
Makes the target session selectable.

2 SET_SESSION_NON_SELECTABLE
Makes the target session non-selectable. A non-selectable session is not selectable from the Shell switch list, nor can the user jump to it via the system hot key. The operator may continue to select a non-selectable windowed session by pressing a mouse button within a visible part of the window.

STATUSDATA Field - BondInd

BondInd (USHORT)
An indicator that specifies which session to bring to the foreground the next time the parent session is selected.

Possible values are shown in the following list:

0 SET_SESSION_UNCHANGED
Leaves the current setting unchanged.

1 SET_SESSION_BOND
Establishes a bond between the parent session and the child session. The child session is brought to the foreground the next time the parent session is selected. If the child session is selected, the child session is brought to the foreground.

2 SET_SESSION_NO_BOND
Specifies bringing the parent session to the foreground the next time the parent session is selected, and bringing the child session to the foreground if the child is selected. Any bond previously established with the child session specified is broken.

STRINGINBUF

The buffer length data structure.

```
typedef struct _STRINGINBUF {
  ULONG     cb;     /*  Length, of the input buffer, in bytes. */
  ULONG     cchIn;  /*  Number of bytes returned. */
} STRINGINBUF;

typedef STRINGINBUF *PSTRINGINBUF;
```

STRINGINBUF Field - cb

cb (ULONG)
Length, of the input buffer, in bytes.

STRINGINBUF Field - cchIn

cchIn (ULONG)
Number of bytes returned.

THRESHOLD

The threshold data structure.

```
typedef struct _THRESHOLD {
  USHORT     Length;    /*  Length field. */
  USHORT     Level1;    /*  First movement level. */
  USHORT     Lev1Mult;  /*  First-level multiplier. */
  USHORT     Level2;    /*  Second movement level. */
  USHORT     Lev2Mult;  /*  Second-level multiplier. */
} THRESHOLD;

typedef THRESHOLD *PTHRESHOLD;
```

THRESHOLD Field - Length

Length (USHORT)
Length field.

THRESHOLD Field - Level1

Level1 (USHORT)
First movement level.

THRESHOLD Field - Lev1Mult

Lev1Mult (USHORT)
First-level multiplier.

THRESHOLD Field - Level2

Level2 (USHORT)
Second movement level.

THRESHOLD Field - Lev2Mult

Lev2Mult (USHORT)
Second-level multiplier.

TIB

Thread Information Block structure.

```
typedef struct _TIB {
  PVOID     tib_pexchain;     /*  Head of exception handler chain. */
  PVOID     tib_pstack;       /*  Pointer to the base of the stack. */
  PVOID     tib_pstacklimit;  /*  Pointer to the end of the stack. */
  PTIB2     tib_ptib2;        /*  Pointer to a system-specific thread information block. */
  ULONG     tib_version;      /*  Version number for this Thread Information Block. */
  ULONG     tib_ordinal;      /*  Thread ordinal number. */
} TIB;

typedef TIB *PTIB;
```

A *thread* is a dispatchable unit of executable code that consists of set of instructions, related CPU register values, and a stack. Every process has at least one thread and can have many threads running at the same time. The application runs when the operating system gives control to a thread in a process. A thread is the basic unit of execution.

Information about a thread is kept in a read/write area of the process address space, called the Thread Information Block (TIB). The operating system creates and maintains a TIB for every thread in the system.

An application can access the TIB of a specific thread using DosGetInfoBlocks.

Note: The fields of this data structure should not be modified unless you are switching stacks, in which case only the *tib_pstack* and *tib_pstacklimit* should changed.

TIB Field - tib_pexchain

tib_pexchain (PVOID)
Head of exception handler chain.

TIB Field - tib_pstack

tib_pstack (PVOID)
Pointer to the base of the stack.

TIB Field - tib_pstacklimit

tib_pstacklimit (PVOID)
Pointer to the end of the stack.

TIB Field - tib_ptib2

tib_ptib2 (PTIB2)
Pointer to a system-specific thread information block.

TIB Field - tib_version

tib_version (ULONG)
Version number for this Thread Information Block.

TIB Field - tib_ordinal

tib_ordinal (ULONG)
Thread ordinal number.

TIB2

System-specific Thread Information Block structure.

```
typedef struct _TIB2 {
  ULONG      tib2_ultid;         /*  Current thread identifier. */
  ULONG      tib2_ulpri;         /*  Current thread priority. */
  ULONG      tib2_version;       /*  Version number for this system-specific Thread Information Block. */
  USHORT     tib2_usMCCount;     /*  Must-complete count. */
  USHORT     tib2_fMCForceFlag;  /*  Must-complete force flag. */
} TIB2;

typedef TIB2 *PTIB2;
```

TIB2 Field - tib2_ultid

tib2_ultid (ULONG)
Current thread identifier.

TIB2 Field - tib2_ulpri

tib2_ulpri (ULONG)
Current thread priority. The low byte of the low word is a hexadecimal value representing a rank (value 0 to 31) within a priority class. Class values, found in the high byte of the low word, are as follows:

0x01
: Idle.

0x02
: Regular.

0x03
: Time critical.

0x04
: Server.

TIB2 Field - tib2_version

tib2_version (ULONG)
Version number for this system-specific Thread Information Block.

TIB2 Field - tib2_usMCCount

tib2_usMCCount (USHORT)
Must-complete count.

TIB2 Field - tib2_fMCForceFlag

tib2_fMCForceFlag (USHORT)
Must-complete force flag.

TID

Thread identity.

```
typedef LHANDLE TID;
```

TRACKLAYOUT

Track layout.

```
typedef struct _TRACKLAYOUT {
  BYTE       bCommand;        /*  Command information. */
  USHORT     usHead;          /*  Physical head on the device that performs the operation. */
  USHORT     usCylinder;      /*  Cylinder to be written, read, or verified. */
  USHORT     usFirstSector;   /*  Logical sector number within the Track Layout Table that starts the I/O operat
  USHORT     cSectors;        /*  Number of sectors to read, write, or verify. */
  struct   {
    USHORT       usSectorNumber;  /*  Sector number. */
    USHORT       usSectorSize;    /*  Sector size. */
  } TrackTable[1];
} TRACKLAYOUT;

typedef TRACKLAYOUT *PTRACKLAYOUT;
```

TRACKLAYOUT Field - bCommand

bCommand (BYTE)
Command information.

Bit 0	If clear (0), the track layout contains nonconsecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

TRACKLAYOUT Field - usHead

usHead (USHORT)
Physical head on the device that performs the operation.

TRACKLAYOUT Field - usCylinder

usCylinder (USHORT)
Cylinder to be written, read, or verified.

TRACKLAYOUT Field - usFirstSector

usFirstSector (USHORT)
Logical sector number within the Track Layout Table that starts the I/O operation.

Note that all the sector numbers start with 0. For example the third sector is number 2.

TRACKLAYOUT Field - cSectors

cSectors (USHORT)
Number of sectors to read, write, or verify.

This value must be less than or equal to the maximum number number of sectors specified in the track table.

The IOCtl subfunctions do not step heads or tracks.

TRACKLAYOUT Field - usSectorNumber

usSectorNumber (USHORT)
Sector number.

There must be one *usSectorNumber* and *usSectorSize* in the track layout table for every sector, up to *cSectors* sectors.

TRACKLAYOUT Field - usSectorSize

usSectorSize (USHORT)
Sector size.

There must be one *usSectorNumber* and *usSectorSize* in the track layout table for every sector, up to *cSectors* sectors.

UCHAR

Single-byte unsigned character or unsigned integer in the range 0 through 255.

```
typedef unsigned char UCHAR;
```

ULONG

32-bit unsigned integer in the range 0 through 4 294 967 295.

```
typedef unsigned long ULONG;
```

USHORT

Unsigned integer in the range 0 through 65 535.

```
typedef unsigned short USHORT;
```

VIOCOLORREG

VIOCOLORREG data structure.

```
typedef struct _VIOCOLORREG {
  USHORT     cb;             /*  Length of this data structure in bytes. */
  USHORT     type;           /*  Request type = 3, get color registers. */
  USHORT     firstcolorreg;  /*  First color register. */
  USHORT     numcolorregs;   /*  Number of color registers to get. */
  PCH        colorregaddr;   /*  Area where the color registers are returned. */
} VIOCOLORREG;

typedef VIOCOLORREG *PVIOCOLORREG;
```

VIOCOLORREG Field - cb

cb (USHORT)
Length of this data structure in bytes.

Length of the structure, including length. The maximum valid value is 12.

VIOCOLORREG Field - type

type (USHORT)
Request type = 3, get color registers.

Request type 3 for color registers.

VIOCOLORREG Field - firstcolorreg

firstcolorreg (USHORT)

First color register.

The first color register to get in the color register sequence. It must be specified in the range 0 through 255. The color registers are returned in sequential order.

VIOCOLORREG Field - numcolorregs

numcolorregs (USHORT)
Number of color registers to get.

The number of color registers to get; must be specified in the range 1 through 256.

VIOCOLORREG Field - colorregaddr

colorregaddr (PCH)
Area where the color registers are returned.

The far address of a data area where the color registers are returned. The size of the data area must be three bytes times the number of color registers to get. The format of each entry returned is as follows:

Byte 1	Red value
Byte 2	Green value
Byte 3	Blue value

VIOCONFIGINFO

VIOCONFIGINFO data structure.

```
typedef struct _VIOCONFIGINFO {
  ULONG     cb;             /*  Length of this data structure */
  ULONG     adapter;        /*  Display adapter type */
  ULONG     display;        /*  Display monitor type */
  ULONG     cbMemory;       /*  Bytes of memory on the adapter */
  ULONG     Configuration;  /*  Configuration number */
  ULONG     VDHVersion;     /*  Reserved */
  ULONG     Flags;
  ULONG     HWBufferSize;   /*  Size to save video state */
  ULONG     FullSaveSize;   /*  Full save size. */
  ULONG     PartSaveSize;   /*  Partial save size */
  ULONG     EMAdaptersOff;  /*  Offset to emulated adapter types */
  ULONG     EMDisplayOFF;   /*  Offset to emulated display types */
} VIOCONFIGINFO;

typedef VIOCONFIGINFO *PVIOCONFIGINFO;
```

VIOCONFIGINFO Field - cb

cb (ULONG)
Length of this data structure

Input parameter to VioGetConfig. The length must be either 4, to indicate the actual length should be returned, or at least 48.

VIOCONFIGINFO Field - adapter

adapter (ULONG)
Display adapter type

Display adapter type.

Value	Definition
0-3	Reserved
3	VGA Display Adapter
4-6	Reserved
7	8514/A
8	Image Adapter/A
9	XGA Display Adapter

Values ranging from 0-4095 are reserved for IBM.

VIOCONFIGINFO Field - display

display (ULONG)
Display monitor type

Display or monitor type.

Value	Definition
0-2	Reserved
3	13-inch Monochrome Display
4	13-inch Color Display
5-8	Reserved
9	16-inch 1024x768 capable color display
10	LCD or Plasma display
11	Large monochrome display
12	14-inch 1024x768 capable color display
13	Reserved

Values ranging from 0-4095 are reserved for IBM.

VIOCONFIGINFO Field - cbMemory

cbMemory (ULONG)
Bytes of memory on the adapter

Amount of memory, in bytes, on the adapter.

VIOCONFIGINFO Field - Configuration

Configuration (ULONG)
Configuration number

Number of the display configuration that this data corresponds to. This is assigned by the video subsystem.

VIOCONFIGINFO Field - VDHVersion

VDHVersion (ULONG)
Reserved

This field is reserved.

VIOCONFIGINFO Field - Flags

Flags (ULONG)

The flag bits are defined as follows:

Bit	Description
31-1	Reserved
0	Power-up display configuration.

VIOCONFIGINFO Field - HWBufferSize

HWBufferSize (ULONG)
Size to save video state

Size of the buffer required by the Base Video Handler (BVH) to save the full hardware state, excluding the physical display buffer.

VIOCONFIGINFO Field - FullSaveSize

FullSaveSize (ULONG)
Full save size.

Maximum size buffer required by the BVH to save the full physical display buffer.

VIOCONFIGINFO Field - PartSaveSize

PartSaveSize (ULONG)
Partial save size

Maximum size buffer required by the BVH to save the portion of the physical display buffer that is overlaid by a pop-up.

VIOCONFIGINFO Field - EMAdaptersOff

EMAdaptersOff (ULONG)
Offset to emulated adapter types

Offset, within the configuration data structure, to the following information describing what other display adapters are emulated by this display adapter.

Number of Data words (ULONG)

Contains a one-word field, specifying a count of data words to follow.

Data word 1 (ULONG)

Bits, set in the data words, identify display adapters emulated. Data word 1 has the following definition:

Bit	Description
0-2	Reserved
3	VGA
4-6	Reserved
7	8514/A Adapter
8	Image Adapter/A
9	XGA Adapter
10-31	Reserved

VIOCONFIGINFO Field - EMDisplayOFF

EMDisplayOFF (ULONG)
Offset to emulated display types

Offset, within the configuration data structure, to the following information describing what other displays are emulated by this display.

Number of Data words (ULONG)

One-word field, specifying a count of data words to follow.

Data word 1 (ULONG)

Bits, set in the data words, identify displays emulated. Data word 1 has the following definition:

Bit	Description
0-2	Reserved
3	13-inch Monochrome Display

4	13-inch Color Display
5-8	Reserved
9	16-inch 1024x768 capable color display
10	LCD or Plasma display
11	Large monochrome display
12	14-inch 1024x768 capable color display
13-31	Reserved

VIOCURSORINFO

VIOCURSORINFO data structure.

```
typedef struct _VIOCURSORINFO {
  USHORT     yStart;  /*  Cursor start line. */
  USHORT     cEnd;    /*  Cursor end line. */
  USHORT     cx;      /*  Cursor width. */
  USHORT     attr;    /*  -1 =hidden cursor. */
} VIOCURSORINFO;

typedef VIOCURSORINFO *PVIOCURSORINFO;
```

VIOCURSORINFO Field - yStart

yStart (USHORT)
Cursor start line.

Horizontal scan line in the character cell that marks the top line of the cursor. If the character cell has n scan lines, 0 is the top scan line of the character cell and (n-1) is the bottom scan line.

VIOCURSORINFO Field - cEnd

cEnd (USHORT)
Cursor end line.

Horizontal scan line in the character cell that marks the bottom line of the cursor. Scan lines within a character cell are numbered as defined in startline.

VIOCURSORINFO Field - cx

cx (USHORT)
Cursor width.

Width of the cursor. In text modes, cursorwidth is the number of columns. The maximum number supported by the OS/2 base video subsystem is 1. In graphics mode, cursorwidth is the number of pels.

VIOCURSORINFO Field - attr

attr (USHORT)
-1 =hidden cursor.

A value of -1 denotes a hidden cursor; all other values in text mode denote normal cursor and in graphics mode denote color attribute.

VIOINTENSITY

VIOINTENSITY data structure.

```
typedef struct _VIOINTENSITY {
  USHORT     cb;     /*  Length of this data structure in bytes. */
  USHORT     type;   /*  Request type = 2, get blink/background intensity switch. */
  USHORT     fs;     /*  Value of blink/background switch. */
} VIOINTENSITY;

typedef VIOINTENSITY *PVIOINTENSITY;
```

VIOINTENSITY Field - cb

cb (USHORT)
Length of this data structure in bytes.

Length of the structure, including length. The only valid value is 6.

VIOINTENSITY Field - type

type (USHORT)
Request type = 2, get blink/background intensity switch.

Request type 2 for blink and background intensity switch.

VIOINTENSITY Field - fs

fs (USHORT)
Value of blink/background switch.

Switch set as:

Value	Definition
0	Blinking foreground colors enabled.
1	High-intensity background colors enabled.

VIOMODEINFO

VIOMODEINFO data structure.

```
typedef struct _VIOMODEINFO {
  USHORT      cb;              /*  Length of this data structure */
  UCHAR       fbType;          /*  Bit mask of mode being set. */
  UCHAR       color;           /*  Number of colors (power of 2). */
  USHORT      col;             /*  The number of text columns. */
  USHORT      row;             /*  The number of text rows. */
  USHORT      hres;            /*  Horizontal resolution. */
  USHORT      vres;            /*  Vertical resolution. */
  UCHAR       fmt_ID;          /*  Attribute format. */
  UCHAR       attrib;          /*  Number of attributes. */
  USHORT      resv;            /*  Reserved. */
  ULONG       buf_addr;        /*  Video aperture address. */
  ULONG       buf_length;      /*  Video aperture length. */
  ULONG       full_length;     /*  Video state full save length. */
  ULONG       partial_length;  /*  Video state partial save length. */
  ULONG       ext_data_addr;   /*  Extra data address. */
} VIOMODEINFO;

typedef VIOMODEINFO *PVIOMODEINFO;
```

VIOMODEINFO Field - cb

cb (USHORT)
Length of this data structure

Input parameter to VioGetMode. Length specifies the length of the data structure in bytes, including length. The value specified on input controls the amount of mode data returned. The minimum structure size required is two bytes. The length is modified on output.

VIOMODEINFO Field - fbType

fbType (UCHAR)
Bit mask of mode being set.

Mode characteristics bit mask:

Bit	Description
7-4	Reserved
3	0 = VGA BIOS compatible modes 1 = Native mode
2	0 = Enable color burst 1 = Disable color burst
1	0 = Text mode 1 = Graphics mode
0	0 = Monochrome compatible mode 1 = Other

VIOMODEINFO Field - color

color (UCHAR)
Number of colors (power of 2).

Number of colors defined as a power of 2. This is equivalent to the number of color bits that define the color, as in this example:

Value	Definition
0	Monochrome
1	2 colors
2	4 colors
4	16 colors
8	256 colors
16	64K colors
24	16M colors

VIOMODEINFO Field - col

col (USHORT)
The number of text columns.

The number of text columns.

VIOMODEINFO Field - row

row (USHORT)
The number of text rows.

The number of text rows.

VIOMODEINFO Field - hres

hres (USHORT)
Horizontal resolution.

Horizontal resolution; the number of pel columns.

VIOMODEINFO Field - vres

vres (USHORT)
Vertical resolution.

Vertical resolution, the number of pel rows.

VIOMODEINFO Field - fmt_ID

fmt_ID (UCHAR)
Attribute format.

The format of the attributes.

VIOMODEINFO Field - attrib

attrib (UCHAR)
Number of attributes.

The number of attributes in a character cell.

VIOMODEINFO Field - resv

resv (USHORT)
Reserved.

VIOMODEINFO Field - buf_addr

buf_addr (ULONG)
Video aperture address.

The physical address of the physical display aperture. This may be zero for emulated video hardware.

VIOMODEINFO Field - buf_length

buf_length (ULONG)
Video aperture length.

The length of the physical display aperture.

VIOMODEINFO Field - full_length

full_length (ULONG)
Video state full save length.

The size of the buffer required for a full save of the video state.

VIOMODEINFO Field - partial_length

partial_length (ULONG)
Video state partial save length.

The size of the buffer required for a partial (pop-up) save of the video state.

VIOMODEINFO Field - ext_data_addr

ext_data_addr (ULONG)
Extra data address.

The virtual address of an extended mode data structure or zero (if none).. The format of the extended mode data structure is determined by the device driver and is unknown to OS/2.

VIOOVERSCAN

VIOOVERSCAN data structure.

```
typedef struct _VIOOVERSCAN {
  USHORT     cb;     /*  Length of this data structure in bytes. */
  USHORT     type;   /*  Request type = 1, get overscan. */
  USHORT     color;  /*  Color value. */
} VIOOVERSCAN;

typedef VIOOVERSCAN *PVIOOVERSCAN;
```

VIOOVERSCAN Field - cb

cb (USHORT)
Length of this data structure in bytes.

Length of the structure, including length. The only valid value is 6.

VIOOVERSCAN Field - type

type (USHORT)
Request type = 1, get overscan.

Request type 1 for overscan (border) color.

VIOOVERSCAN Field - color

color (USHORT)
Color value.

The color value.

VIOPALSTATE

VIOPALSTATE data structure.

```
typedef struct _VIOPALSTATE {
  USHORT     cb;          /*  Length of this data structure in bytes. */
  USHORT     type;        /*  Request type = 0, get palette registers. */
  USHORT     iFirst;      /*  First palette register to return. */
  USHORT     acolor[1];   /*  Color value palette register. */
} VIOPALSTATE;

typedef VIOPALSTATE *PVIOPALSTATE;
```

VIOPALSTATE Field - cb

cb (USHORT)
Length of this data structure in bytes.

Length of the structure, including length. The maximum valid value is 38.

VIOPALSTATE Field - type

type (USHORT)
Request type = 0, get palette registers.

Request type 0 for palette registers.

VIOPALSTATE Field - iFirst

iFirst (USHORT)
First palette register to return.

The first palette register in the palette register sequence. It must be specified in the range 0 through 15. The palette registers are returned in sequential order. The number returned is based upon length.

VIOPALSTATE Field - acolor[1]

acolor[1] (USHORT)
Color value palette register.

The color value for each palette register. The maximum number of entries in the color value array is 16.

VIOSETTARGET

VIOSETTARGET data structure.

```
typedef struct _VIOSETTARGET {
  USHORT     cb;                /*  Length of this data structure in bytes. */
  USHORT     type;              /*  Request type = 6, get display configuration. */
  USHORT     defaultalgorithm;
} VIOSETTARGET;

typedef VIOSETTARGET *PVIOSETTARGET;
```

VIOSETTARGET Field - cb

cb (USHORT)
Length of this data structure in bytes.

Length of the structure, including length. The only valid value is 6.

VIOSETTARGET Field - type

type (USHORT)
Request type = 6, get display configuration.

Request type 6 to get display configuration selected to be the target of the next VioSetMode.

VIOSETTARGET Field - defaultalgorithm

defaultalgorithm (USHORT)

Configuration:

Value	Definition
0	Default selection algorithm. See VioSetMode.
1	Primary
2	Secondary

VIOSETULINELOC

VIOSETULINELOC data structure.

```
typedef struct _VIOSETULINELOC {
  USHORT     cb;         /*  Length of this data structure in bytes. */
  USHORT     type;       /*  Request type = 5, get scan line for underlining. */
  USHORT     scanline;
} VIOSETULINELOC;

typedef VIOSETULINELOC *PVIOSETULINELOC;
```

VIOSETULINELOC Field - cb

cb (USHORT)

Length of this data structure in bytes.

Length of the structure, including length. The only valid value is 6.

VIOSETULINELOC Field - type

type (USHORT)

Request type = 5, get scan line for underlining.

Request type 5 to get the scan line for underlining.

VIOSETULINELOC Field - scanline

scanline (USHORT)

The value returned is in the range 0 through 31 and is the scan line minus 1. A value of 32 means underlining is disabled.

VOID

A data area of undefined format.

```
#define VOID void
```

VOLUMELABEL

Volume label structure.

```
typedef struct _VOLUMELABEL {
  BYTE     cch;            /*  Length of the volume label, not including the null. */
  CHAR     szVolLabel[12];  /*  Volume label. */
} VOLUMELABEL;

typedef VOLUMELABEL *PVOLUMELABEL;
```

VOLUMELABEL Field - cch

cch (BYTE)
Length of the volume label, not including the null.

VOLUMELABEL Field - szVolLabel[12]

szVolLabel[12] (CHAR)
Volume label.

This is an ASCIIZ string.

Errors

For errors 0 through 499, the following shows the numerical value of an error, its symbolic name, and a brief description of the error. For errors 500 and above, the following shows the numerical value of an error and its symbolic name.

Select a range of Control Program error return codes:

0 to 99

100 to 199

200 to 299

300 to 399

400 to 499

500 to 599

600 to 1999

2000 to 59999

60000 to 65079

Errors 0 to 99

The following shows the numerical value of an error, its symbolic name, and a brief description of the error.

0 NO_ERROR
No error occurred.

1 ERROR_INVALID_FUNCTION
Function number is not valid.

2 ERROR_FILE_NOT_FOUND
File not found.

3 ERROR_PATH_NOT_FOUND
Path not found.

4 ERROR_TOO_MANY_OPEN_FILES
Too many open files (no handles left).

5 ERROR_ACCESS_DENIED
Access denied.

6 ERROR_INVALID_HANDLE
Handle is not valid.

7 ERROR_ARENA_TRASHED
Memory control blocks destroyed.

8 ERROR_NOT_ENOUGH_MEMORY
Insufficient memory.

9 ERROR_INVALID_BLOCK
Memory-block address is not valid.

10 ERROR_BAD_ENVIRONMENT
Environment is not valid.

11 ERROR_BAD_FORMAT
Format is not valid.

12 ERROR_INVALID_ACCESS
Access code is not valid.

13 ERROR_INVALID_DATA
Data is not valid.

14 Reserved.

15 ERROR_INVALID_DRIVE
Drive is not valid.

16 ERROR_CURRENT_DIRECTORY
Attempting to remove current directory.

17 ERROR_NOT_SAME_DEVICE
Not same device.

18 ERROR_NO_MORE_FILES
No more files.

19 ERROR_WRITE_PROTECT
Attempt to write on write-protected diskette.

20 ERROR_BAD_UNIT
Unknown unit.

21 ERROR_NOT_READY
Drive not ready.

22 ERROR_BAD_COMMAND
Unknown command.

23 ERROR_CRC
Data error - cyclic redundancy check.

24 ERROR_BAD_LENGTH
Request structure length is not valid.

25 ERROR_SEEK
Seek error.

26 ERROR_NOT_DOS_DISK
Unknown media type.

27 ERROR_SECTOR_NOT_FOUND
Sector not found.

28 ERROR_OUT_OF_PAPER
Printer is out of paper.

29 ERROR_WRITE FAULT
Write fault.

30 ERROR_READ_FAULT
Read fault.

31 ERROR_GEN_FAILURE
General failure. For DosGetShrSeg and DosGetNamedSharedMem, indicates that a segment's maximum reference count of 65535 has been exceeded.

32 ERROR_SHARING_VIOLATION
Sharing violation.

33 ERROR_LOCK_VIOLATION
Lock violation.

34 ERROR_WRONG_DISK
Disk change is not valid.

35 ERROR_FCB_UNAVAILABLE
FCB unavailable.

36 ERROR_SHARING_BUFFER_EXCEEDED
Sharing buffer overflow.

37 ERROR_CODE_PAGE_MISMATCHED
Code page does not match.

38 ERROR_HANDLE_EOF
End of file reached.

39 ERROR_HANDLE_DISK_FULL
Disk is full.

40-49 Reserved.

50 ERROR_NOT_SUPPORTED
Network request not supported.

51 ERROR_REM_NOT_LIST
Remote network node is not online.

52 ERROR_DUP_NAME
Duplicate file name in network.

53 ERROR_BAD_NETPATH
Network path not found.

54 ERROR_NETWORK_BUSY
Network is busy.

55 ERROR_DEV_NOT_EXIST
Device is not installed in network.

56 ERROR_TOO_MANY_CMDS
Network command limit reached.

57 ERROR_ADAP_HDW_ERR
Network adapter hardware error.

58 ERROR_BAD_NET_RESP
Incorrect response in network.

59 ERROR_UNEXP_NET_ERR
Unexpected error in network.

60 ERROR_BAD_REM_ADAP
Remote network adapter error.

61 ERROR_PRINTQ_FULL
Network printer queue is full.

62 ERROR_NO_SPOOL_SPACE
No space in print spool file.

63 ERROR_PRINT_CANCELLED
Print spool file deleted.

64 ERROR_NETNAME_DELETED
Network name deleted.

65 ERROR_NETWORK_ACCESS_DENIED
Access to network denied.

66 ERROR_BAD_DEV_TYPE
Device type for network is not valid.

67 ERROR_BAD_NET_NAME
Network name not found.

68 ERROR_TOO_MANY_NAMES
Network name limit exceeded.

69 ERROR_TOO_MANY_SESS
Network session limit exceeded.

70 ERROR_SHARING_PAUSED
Temporary pause in network.

71 ERROR_REQ_NOT_ACCEP
Network request denied.

72 ERROR_REDIR_PAUSED
Pause in network print disk redirection.

73 ERROR_SBCS_ATT_WRITE_PROT
Attempted write on protected disk.

74 ERROR_SBCS_GENERAL_FAILURE
General failure, single-byte character set.

75 ERROR_XGA_OUT_MEMORY
XGA is out of memory.

76-79 Reserved.

80 ERROR_FILE_EXISTS
File exists.

81 ERROR_DUP_FCB
Reserved.

82 ERROR_CANNOT_MAKE
Cannot make directory entry.

83 ERROR_FAIL_I24
Failure on INT 24.

84 ERROR_OUT_OF_STRUCTURES
Too many redirections.

85 ERROR_ALREADY_ASSIGNED
Duplicate redirection.

86 ERROR_INVALID_PASSWORD
Password is not valid.

87 ERROR_INVALID_PARAMETER
Parameter is not valid.

88 ERROR_NET_WRITE_FAULT
Network device fault.

89 ERROR_NO_PROC_SLOTS
No process slots available.

90 ERROR_NOT_FROZEN
System error.

90 ERROR_SYS_COMP_NOT_LOADED
System error.

91 ERR_TSTOVFL
Timer service table overflow.

92 ERR_TSTDUP
Timer service table duplicate.

93 ERROR_NO_ITEMS
No items to work on.

95 ERROR_INTERRUPT
Interrupted system call.

99 ERROR_DEVICE_IN_USE
Device in use.

Errors 100 to 199

The following shows the numerical value of an error, its symbolic name, and a brief description of the error.

100 ERROR_TOO_MANY_SEMAPHORES
User/system open semaphore limit reached.

101 ERROR_EXCL_SEM_ALREADY_OWNED
Exclusive semaphore already owned.

102 ERROR_SEM_IS_SET
DosCloseSem found semaphore set.

103 ERROR_TOO_MANY_SEM_REQUESTS
Too many exclusive semaphore requests.

104 ERROR_INVALID_AT_INTERRUPT_TIME
Operation at interrupt time is not valid.

105 ERROR_SEM_OWNER_DIED
Previous semaphore owner ended without freeing semaphore.

106 ERROR_SEM_USER_LIMIT
Semaphore limit exceeded.

107 ERROR_DISK_CHANGE
Insert drive B disk into drive A.

108 ERROR_DRIVE_LOCKED
Drive locked by another process.

109 ERROR_BROKEN_PIPE
Write on pipe with no reader.

110 ERROR_OPEN_FAILED
Open/create failed due to explicit fail command.

111 ERROR_BUFFER_OVERFLOW
Buffer passed to system call too small to hold return data.

112 ERROR_DISK_FULL
Not enough space on the disk.

113 ERROR_NO_MORE_SEARCH_HANDLES
Cannot allocate another search structure and handle.

114 ERROR_INVALID_TARGET_HANDLE
Target handle in DosDupHandle is not valid.

115 ERROR_PROTECTION_VIOLATION
User virtual address is not valid.

116 ERROR_VIOKBD_REQUEST
Error on display write or keyboard read.

117 ERROR_INVALID_CATEGORY
Category for DevIOCtl not defined.

118 ERROR_INVALID_VERIFY_SWITCH
Value passed for verify flag is not valid.

119 ERROR_BAD_DRIVER_LEVEL
Level four driver not found.

120 ERROR_CALL_NOT_IMPLEMENTED
Function called is not valid.

121 ERROR_SEM_TIMEOUT
Time-out occurred from semaphore API function.

122 ERROR_INSUFFICIENT_BUFFER
Data buffer too small.

123 ERROR_INVALID_NAME
Illegal character or file-system name is not valid.

124 ERROR_INVALID_LEVEL
Level for information retrieval or setting is not valid.

125 ERROR_NO_VOLUME_LABEL
No volume label found with DosQueryFSInfo function.

126 ERROR_MOD_NOT_FOUND
Module handle not found with DosQueryProcAddr(), DosQueryModAddr().

127 ERROR_PROC_NOT_FOUND
Procedure address not found with DosQueryProcAddr().

128 ERROR_WAIT_NO_CHILDREN
DosWaitChild finds no children.

129 ERROR_CHILD_NOT_COMPLETE

DosWaitChild children not ended.

130 ERROR_DIRECT_ACCESS_HANDLE
Handle operation is not valid for direct disk-access handles.

131 ERROR_NEGATIVE_SEEK
Attempting seek to negative offset.

132 ERROR_SEEK_ON_DEVICE
Application trying to seek on device or pipe.

133 ERROR_IS_JOIN_TARGET
Drive has previously joined drives.

134 ERROR_IS_JOINED
Drive is already joined.

135 ERROR_IS_SUBSTED
Drive is already substituted.

136 ERROR_NOT_JOINED
Cannot delete drive that is not joined.

137 ERROR_NOT_SUBSTED
Cannot delete drive that is not substituted.

138 ERROR_JOIN_TO_JOIN
Cannot join to a joined drive.

139 ERROR_SUBST_TO_SUBST
Cannot substitute to a substituted drive.

140 ERROR_JOIN_TO_SUBST
Cannot join to a substituted drive.

141 ERROR_SUBST_TO_JOIN
Cannot substitute to a joined drive.

142 ERROR_BUSY_DRIVE
Specified drive is busy.

143 ERROR_SAME_DRIVE
Cannot join or substitute a drive to a directory on the same drive.

144 ERROR_DIR_NOT_ROOT
Directory must be a subdirectory of the root.

145 ERROR_DIR_NOT_EMPTY
Directory must be empty to use join command.

146 ERROR_IS_SUBST_PATH
Path specified is being used in a substitute.

147 ERROR_IS_JOIN_PATH
Path specified is being used in a join.

148 ERROR_PATH_BUSY
Path specified is being used by another process.

149 ERROR_IS_SUBST_TARGET
Cannot join or substitute a drive that has a directory that is the target of a previous substitute.

150 ERROR_SYSTEM_TRACE
System trace error.

151 ERROR_INVALID_EVENT_COUNT
DosWaitMuxWaitSem errors.

152 ERROR_TOO_MANY_MUXWAITERS
System limit of 100 entries reached.

153 ERROR_INVALID_LIST_FORMAT
List format is not valid.

154 ERROR_LABEL_TOO_LONG
Volume label too big.

155 ERROR_TOO_MANY_TCBS
Cannot create another TCB.

156 ERROR_SIGNAL_REFUSED
Signal refused.

157 ERROR_DISCARDED
Segment is discarded.

158 ERROR_NOT_LOCKED
Segment is not locked.

159 ERROR_BAD_THREADID_ADDR
Thread-identity address is not valid.

160 ERROR_BAD_ARGUMENTS
Environment pointer is not valid.

161 ERROR_BAD_PATHNAME
Path name is not valid.

162 ERROR_SIGNAL_PENDING
Signal already pending.

163 ERROR_UNCERTAIN_MEDIA
Error with INT 24 mapping.

164 ERROR_MAX_THRDS_REACHED
No more process slots.

165 ERROR_MONITORS_NOT_SUPPORTED
Error with INT 24 mapping.

166 ERROR_UNC_DRIVER_NOT_INSTALLED
Default redirection return code.

167 ERROR_LOCK_FAILED
Locking failed.

168 ERROR_SWAPIO_FAILED
Swap I/O failed.

169 ERROR_SWAPIN_FAILED
Swap in failed.

170 ERROR_BUSY
Segment is busy.

171-172 Reserved.

173 ERROR_CANCEL_VIOLATION
A lock request is not outstanding for the specified file range, or the range length is zero.

174 ERROR_ATOMIC_LOCK_NOT_SUPPORTED
The file-system driver (FSD) does not support atomic lock operations. Versions of OS/2 prior to version 2.00 do not support atomic lock operations.

175 ERROR_READ_LOCKS_NOT_SUPPORTED
The file system driver (FSD) does not support shared read locks.

176-179 Reserved.

180 ERROR_INVALID_SEGMENT_NUMBER
Segment number is not valid.

181 ERROR_INVALID_CALLGATE
Call gate is not valid.

182 ERROR_INVALID_ORDINAL
Ordinal is not valid.

183 ERROR_ALREADY_EXISTS
Shared segment already exists.

184 ERROR_NO_CHILD_PROCESS
No child process to wait for.

185 ERROR_CHILD_ALIVE_NOWAIT
NoWait specified and child alive.

186 ERROR_INVALID_FLAG_NUMBER
Flag number is not valid.

187 ERROR_SEM_NOT_FOUND
Semaphore does not exist.

188 ERROR_INVALID_STARTING_CODESEG
Starting code segment is not valid, incorrect END (label) directive.

189 ERROR_INVALID_STACKSEG
Stack segment is not valid.

190 ERROR_INVALID_MODULETYPE
Module type is not valid - dynamic-link library file cannot be used as an application. Application cannot be used as a dynamic-link library.

191 ERROR_INVALID_EXE_SIGNATURE
EXE signature is not valid - file is a DOS mode program or an improper program.

192 ERROR_EXE_MARKED_INVALID
EXE marked is not valid - link detected errors when the application was created.

193 ERROR_BAD_EXE_FORMAT
EXE format not valid - file is a DOS mode program or an improper program.

194 ERROR_ITERATED_DATA_EXCEEDS_64k
Iterated data exceeds 64KB - there is more than 64KB of data in one of the segments of the file.

195 ERROR_INVALID_MINALLOCSIZE
Minimum allocation size is not valid - the size is specified to be less than the size of the segment data in the file.

196 ERROR_DYNLINK_FROM_INVALID_RING
Dynamic link from privilege level is not valid - privilege level 2 routine cannot link to dynamic-link libraries.

197 ERROR_IOPL_NOT_ENABLED
IOPL not enabled - IOPL set to NO in CONFIG.SYS.

198 ERROR_INVALID_SEGDPL
Segment descriptor privilege level is not valid - can only have privilege levels of 2 and 3.

199 ERROR_AUTODATASEG_EXCEEDS_64k
Automatic data segment exceeds 64KB.

Errors 200 to 299

The following shows the numerical value of an error, its symbolic name, and a brief description of the error.

200 ERROR_RING2SEG_MUST_BE_MOVABLE
Privilege level 2 segment must be movable.

201 ERROR_RELOC_CHAIN_XEEDS_SEGLIM
Relocation chain exceeds segment limit.

202 ERROR_INFLOOP_IN_RELOC_CHAIN
Infinite loop in relocation chain segment.

203 ERROR_ENVVAR_NOT_FOUND
Environment variable not found.

204 ERROR_NOT_CURRENT_CTRY
Not current country.

205 ERROR_NO_SIGNAL_SENT

No signal sent - no process in the command subtree has a signal handler.

206 ERROR_FILENAME_EXCED_RANGE
File name or extension is greater than 8.3 characters.

207 ERROR_RING2_STACK_IN_USE
Privilege level 2 stack is in use.

208 ERROR_META_EXPANSION_TOO_LONG
Meta (global) expansion is too long.

209 ERROR_INVALID_SIGNAL_NUMBER
Signal number is not valid.

210 ERROR_THREAD_1_INACTIVE
Inactive thread.

211 ERROR_INFO_NOT_AVAIL
File system information is not available for this file.

212 ERROR_LOCKED
Locked error.

213 ERROR_BAD_DYNALINK
Attempted to execute a non-family API in DOS mode.

214 ERROR_TOO_MANY_MODULES
Too many modules.

215 ERROR_NESTING_NOT_ALLOWED
Nesting is not allowed.

216 ERROR_CANNOT_SHRINK
System error.

217 ERROR_ZOMBIE_PROCESS
Zombie process.

218 ERROR_STACK_IN_HIGH_MEMORY
Stack is in high memory.

219 ERROR_INVALID_EXITROUTINE_RING
Exit routine ring is not valid.

220 ERROR_GETBUF_FAILED
Get buffer failed.

221 ERROR_FLUSHBUF_FAILED
Flush buffer failed.

222 ERROR_TRANSFER_TOO_LONG
Transfer is too long.

223 ERROR_FORCENOSWAP_FAILED
System error.

224 ERROR_SMG_NO_TARGET_WINDOW
The application window was created without the FCF_TASKLIST style, or the application window not yet been created or has already been destroyed.

228 ERROR_NO_CHILDREN
No child process.

229 ERROR_INVALID_SCREEN_GROUP
Session is not valid.

230 ERROR_BAD_PIPE
The pipe is non-existent pipe or the operation is not valid.

231 ERROR_PIPE_BUSY
Pipe is busy.

232 ERROR_NO_DATA
No data available on non-blocking read.

233 ERROR_PIPE_NOT_CONNECTED

Pipe was disconnected by server.

234 ERROR_MORE_DATA
More data is available.

240 ERROR_VC_DISCONNECTED
Session was dropped due to errors.

250 ERROR_CIRCULARITY_REQUESTED
Renaming a directory that would cause a circularity problem.

251 ERROR_DIRECTORY_IN_CDS
Renaming a directory that is in use.

252 ERROR_INVALID_FSD_NAME
Trying to access an FSD that is not valid.

253 ERROR_INVALID_PATH
Pseudo device is not valid.

254 ERROR_INVALID_EA_NAME
Character in name or cbName is not valid.

255 ERROR_EA_LIST_INCONSISTENT
List does not match its size, or there are EAs that are not valid in the list.

256 ERROR_EA_LIST_TOO_LONG
FEAList is longer than 64K-1 bytes.

257 ERROR_NO_META_MATCH
String does not match expression.

258 ERROR_FINDNOTIFY_TIMEOUT
System error.

259 ERROR_NO_MORE_ITEMS
DosQueryFSAttach ordinal query.

260 ERROR_SEARCH_STRUC_REUSED
DOS mode findfirst/next search structure reused.

261 ERROR_CHAR_NOT_FOUND
Character not found.

262 ERROR_TOO_MUCH_STACK
Stack request exceeds system limit.

263 ERROR_INVALID_ATTR
Attribute is not valid.

264 ERROR_INVALID_STARTING_RING
Starting ring is not valid.

265 ERROR_INVALID_DLL_INIT_RING
DLL INIT ring is not valid.

266 ERROR_CANNOT_COPY
Cannot copy.

267 ERROR_DIRECTORY
Used by DOSCOPY in doscall1.

268 ERROR_OPLOCKED_FILE
Oplocked file.

269 ERROR_OPLOCK_THREAD_EXISTS
Oplock thread exists.

270 ERROR_VOLUME_CHANGED
Volume changed.

271 ERROR_FINDNOTIFY_HANDLE_IN_USE
Handle in use.

272 ERROR_FINDNOTIFY_HANDLE_CLOSED
Handle closed.

273 ERROR_NOTIFY_OBJECT_REMOVED
Object removed.

274 ERROR_ALREADY_SHUTDOWN
System is already shut down.

275 ERROR_EAS_DIDNT_FIT
Buffer is not big enough to hold the EAs.

276 ERROR_EA_FILE_CORRUPT
EA file has been damaged.

277 ERROR_EA_TABLE_FULL
EA table is full.

278 ERROR_INVALID_EA_HANDLE
EA handle is not valid.

279 ERROR_NO_CLUSTER
No cluster.

280 ERROR_CREATE_EA_FILE
Cannot create the EA file.

281 ERROR_CANNOT_OPEN_EA_FILE
Cannot open the EA file.

282 ERROR_EAS_NOT_SUPPORTED
Destination file system does not support EAs.

283 ERROR_NEED_EAS_FOUND
Destination file system does not support EAs, and the source file's EAs contain a need EA.

284 ERROR_DUPLICATE_HANDLE
The handle already exists.

285 ERROR_DUPLICATE_NAME
The name already exists.

286 ERROR_EMPTY_MUXWAIT
The list of semaphores in a muxwait semaphore is empty.

287 ERROR_MUTEX_OWNED
The calling thread owns one or more of the mutex semaphores in the list.

288 ERROR_NOT_OWNER
Caller does not own the semaphore.

289 ERROR_PARAM_TOO_SMALL
Parameter is not large enough to contain all of the semaphore records in the muxwait semaphore.

290 ERROR_TOO_MANY_HANDLES
Limit reached for number of handles.

291 ERROR_TOO_MANY_OPENS
There are too many files or semaphores open.

292 ERROR_WRONG_TYPE
Attempted to create wrong type of semaphore.

293 ERROR_UNUSED_CODE
Code is not used.

294 ERROR_THREAD_NOT_TERMINATED
Thread has not ended.

295 ERROR_INIT_ROUTINE_FAILED
Initialization routine failed.

296 ERROR_MODULE_IN_USE
Module is in use.

297 ERROR_NOT_ENOUGH_WATCHPOINTS
There are not enough watchpoints.

298 ERROR_TOO_MANY_POSTS
Post count limit was reached for an event semaphore.

299 ERROR_ALREADY_POSTED
Event semaphore is already posted.

Errors 300 to 399

The following shows the numerical value of an error, its symbolic name, and a brief description of the error.

300 ERROR_ALREADY_RESET
Event semaphore is already reset.

301 ERROR_SEM_BUSY
Semaphore is busy.

302 Reserved

303 ERROR_INVALID_PROCID
Process identity is not valid.

304 ERROR_INVALID_PDELTA
Priority delta is not valid.

305 ERROR_NOT_DESCENDANT
Not descendant.

306 ERROR_NOT_SESSION_MANAGER
Requestor not session manager.

307 ERROR_INVALID_PCLASS
P class is not valid.

308 ERROR_INVALID_SCOPE
Scope is not valid.

309 ERROR_INVALID_THREADID
Thread identity is not valid.

310 ERROR_DOSSUB_SHRINK
Cannot shrink segment - DosSubSetMem.

311 ERROR_DOSSUB_NOMEM
No memory to satisfy request - DosSubAllocMem.

312 ERROR_DOSSUB_OVERLAP
Overlap of the specified block with a block of allocated memory - DosSubFreeMem.

313 ERROR_DOSSUB_BADSIZE
Size parameter is not valid - DosSubAllocMem or DosSubFreeMem.

314 ERROR_DOSSUB_BADFLAG
Flag parameter is not valid - DosSubSetMem.

315 ERROR_DOSSUB_BADSELECTOR
Segment selector is not valid.

316 ERROR_MR_MSG_TOO_LONG
Message too long for buffer.

317 ERROR_MR_MID_NOT_FOUND
Message identity number not found.

318 ERROR_MR_UN_ACC_MSGF
Unable to access message file.

319 ERROR_MR_INV_MSGF_FORMAT
Message file format is not valid.

320 ERROR_MR_INV_IVCOUNT
Insertion variable count is not valid.

321 ERROR_MR_UN_PERFORM
Unable to perform function.

322 ERROR_TS_WAKEUP
Unable to wake up.

323 ERROR_TS_SEMHANDLE
System semaphore is not valid.

324 ERROR_TS_NOTIMER
No timers available.

326 ERROR_TS_HANDLE
Timer handle is not valid.

327 ERROR_TS_DATETIME
Date or time is not valid.

328 ERROR_SYS_INTERNAL
Internal system error.

329 ERROR_QUE_CURRENT_NAME
Current queue name does not exist.

330 ERROR_QUE_PROC_NOT_OWNED
Current process does not own queue.

331 ERROR_QUE_PROC_OWNED
Current process owns queue.

332 ERROR_QUE_DUPLICATE
Duplicate queue name.

333 ERROR_QUE_ELEMENT_NOT_EXIST
Queue element does not exist.

334 ERROR_QUE_NO_MEMORY
Inadequate queue memory.

For DosOpenQueue, DosCreateQueue, and DosWriteQueue, the following applies: These calls use a system-wide pool of memory. Every DosOpenQueue and DosCreateQueue uses up 34 bytes of memory, which is freed on close. Every DosWriteQueue uses 24 bytes of memory, which is freed on read. If too many elements are written to queues, further opens, creates, reads, or writes fail with this error code.

335 ERROR_QUE_INVALID_NAME
Queue name is not valid.

336 ERROR_QUE_INVALID_PRIORITY
Queue priority parameter is not valid.

337 ERROR_QUE_INVALID_HANDLE
Queue handle is not valid.

338 ERROR_QUE_LINK_NOT_FOUND
Queue link not found.

339 ERROR_QUE_MEMORY_ERROR
Queue memory error.

340 ERROR_QUE_PREV_AT_END
Previous queue element was at end of queue.

341 ERROR_QUE_PROC_NO_ACCESS
Process does not have access to queues.

342 ERROR_QUE_EMPTY
Queue is empty.

343 ERROR_QUE_NAME_NOT_EXIST
Queue name does not exist.

344 ERROR_QUE_NOT_INITIALIZED
Queues not initialized.

345 ERROR_QUE_UNABLE_TO_ACCESS
Unable to access queues.

346 ERROR_QUE_UNABLE_TO_ADD
Unable to add new queue.

347 ERROR_QUE_UNABLE_TO_INIT
Unable to initialize queues.

349 ERROR_VIO_INVALID_MASK
Function replaced is not valid.

350 ERROR_VIO_PTR
Pointer to parameter is not valid.

351 ERROR_VIO_APTR
Pointer to attribute is not valid.

352 ERROR_VIO_RPTR
Pointer to row is not valid.

353 ERROR_VIO_CPTR
Pointer to column is not valid.

354 ERROR_VIO_LPTR
Pointer to length is not valid.

355 ERROR_VIO_MODE
Unsupported screen mode.

356 ERROR_VIO_WIDTH
Cursor width value is not valid.

357 ERROR_VIO_ATTR
Cursor attribute value is not valid.

358 ERROR_VIO_ROW
Row value is not valid.

359 ERROR_VIO_COL
Column value is not valid.

360 ERROR_VIO_TOPROW
TopRow value is not valid.

361 ERROR_VIO_BOTROW
BotRow value is not valid.

362 ERROR_VIO_RIGHTCOL
Right column value is not valid.

363 ERROR_VIO_LEFTCOL
Left column value is not valid.

364 ERROR_SCS_CALL
Call issued by other than session manager.

365 ERROR_SCS_VALUE
Value is not for save or restore.

366 ERROR_VIO_WAIT_FLAG
Wait flag setting is not valid.

367 ERROR_VIO_UNLOCK
Screen not previously locked.

368 ERROR_SGS_NOT_SESSION_MGR
Caller not session manager.

369 ERROR_SMG_INVALID_SGID
Session identity is not valid.

369 ERROR_SMG_INVALID_SESSION_ID
Session ID is not valid.

370 ERROR_SMG_NOSG
No sessions available.

370 ERROR_SMG_NO_SESSIONS
No sessions available.

371 ERROR_SMG_GRP_NOT_FOUND
Session not found.

371 ERROR_SMG_SESSION_NOT_FOUND
Session not found.

372 ERROR_SMG_SET_TITLE
Title sent by shell or parent cannot be changed.

373 ERROR_KBD_PARAMETER
Parameter to keyboard is not valid.

374 ERROR_KBD_NO_DEVICE
No device.

375 ERROR_KBD_INVALID_IOWAIT
I/O wait specified is not valid.

376 ERROR_KBD_INVALID_LENGTH
Length for keyboard is not valid.

377 ERROR_KBD_INVALID_ECHO_MASK
Echo mode mask is not valid.

378 ERROR_KBD_INVALID_INPUT_MASK
Input mode mask is not valid.

379 ERROR_MON_INVALID_PARMS
One or more parameters to DosMon is not valid.

380 ERROR_MON_INVALID_DEVNAME
Device name string is not valid.

381 ERROR_MON_INVALID_HANDLE
Device handle is not valid.

382 ERROR_MON_BUFFER_TOO_SMALL
Buffer too small.

383 ERROR_MON_BUFFER_EMPTY
Buffer is empty.

384 ERROR_MON_DATA_TOO_LARGE
Data record is too large.

385 ERROR_MOUSE_NO_DEVICE
Mouse device closed; the device handle is not valid.

386 ERROR_MOUSE_INV_HANDLE
Mouse device closed; the device handle is not valid.

387 ERROR_MOUSE_INV_PARMS
Parameters for display mode are not valid.

388 ERROR_MOUSE_CANT_RESET
Function assigned and cannot be reset.

389 ERROR_MOUSE_DISPLAY_PARMS
Parameters for display mode are not valid.

390 ERROR_MOUSE_INV_MODULE
Module not valid.

391 ERROR_MOUSE_INV_ENTRY_PT
Entry point not valid.

392 ERROR_MOUSE_INV_MASK
Function mask is not valid.

393 NO_ERROR_MOUSE_NO_DATA

No valid data.

394 NO_ERROR_MOUSE_PTR_DRAWN
Pointer drawn.

395 ERROR_INVALID_FREQUENCY
Frequency for beep is not valid.

396 ERROR_NLS_NO_COUNTRY_FILE
Cannot find COUNTRY.SYS file.

397 ERROR_NLS_OPEN_FAILED
Cannot open COUNTRY.SYS file.

398 ERROR_NLS_NO_CTRY_CODE
Country code not found.

398 ERROR_NO_COUNTRY_OR_CODEPAGE
Country code not found.

399 ERROR_NLS_TABLE_TRUNCATED
Table returned information truncated, buffer is too small.

Errors 400 to 499

The following shows the numerical value of an error, its symbolic name, and a brief description of the error.

400 ERROR_NLS_BAD_TYPE
Selected type does not exist.

401 ERROR_NLS_TYPE_NOT_FOUND
Selected type is not in file.

402 ERROR_VIO_SMG_ONLY
Valid from session manager only.

403 ERROR_VIO_INVALID_ASCIIZ
ASCIIZ length is not valid.

404 ERROR_VIO_DEREGISTER
VioDeRegister not allowed.

405 ERROR_VIO_NO_POPUP
Pop-up window not allocated.

406 ERROR_VIO_EXISTING_POPUP
Pop-up window on screen (NoWait).

407 ERROR_KBD_SMG_ONLY
Valid from session manager only.

408 ERROR_KBD_INVALID_ASCIIZ
ASCIIZ length is not valid.

409 ERROR_KBD_INVALID_MASK
Replacement mask is not valid.

410 ERROR_KBD_REGISTER
KbdRegister not allowed.

411 ERROR_KBD_DEREGISTER
KbdDeRegister not allowed.

412 ERROR_MOUSE_SMG_ONLY
Valid from session manager only.

413 ERROR_MOUSE_INVALID_ASCIIZ
ASCIIZ length is not valid.

414 ERROR_MOUSE_INVALID_MASK

Replacement mask is not valid.

415 ERROR_MOUSE_REGISTER
Mouse register not allowed.

416 ERROR_MOUSE_DEREGISTER
Mouse deregister not allowed.

417 ERROR_SMG_BAD_ACTION
Action specified is not valid.

418 ERROR_SMG_INVALID_CALL
INIT called more than once, or the session identity is not valid.

419 ERROR_SCS_SG_NOTFOUND
New session number.

420 ERROR_SCS_NOT_SHELL
Caller is not shell.

421 ERROR_VIO_INVALID_PARMS
Parameters passed are not valid.

422 ERROR_VIO_FUNCTION_OWNED
Save/restore already owned.

423 ERROR_VIO_RETURN
Non-destruct return (undo).

424 ERROR_SCS_INVALID_FUNCTION
Caller function is not valid.

425 ERROR_SCS_NOT_SESSION_MGR
Caller not session manager.

426 ERROR_VIO_REGISTER
Vio register not allowed.

427 ERROR_VIO_NO_MODE_THREAD
No mode restore thread in SG.

428 ERROR_VIO_NO_SAVE_RESTORE_THD
No save/restore thread in SG.

429 ERROR_VIO_IN_BG
Function in background is not valid.

430 ERROR_VIO_ILLEGAL_DURING_POPUP
Function not allowed during pop-up window.

431 ERROR_SMG_NOT_BASESHELL
Caller is not the base shell.

432 ERROR_SMG_BAD_STATUSREQ
Status requested is not valid.

433 ERROR_QUE_INVALID_WAIT
NoWait parameter out of bounds.

434 ERROR_VIO_LOCK
Error returned from Scroll Lock.

435 ERROR_MOUSE_INVALID_IOWAIT
Parameters for IOWait are not valid.

436 ERROR_VIO_INVALID_HANDLE
VIO handle is not valid.

437 ERROR_VIO_ILLEGAL_DURING_LOCK
Function not allowed during screen lock.

438 ERROR_VIO_INVALID_LENGTH
VIO length is not valid.

439 ERROR_KBD_INVALID_HANDLE
KBD handle is not valid.

440 ERROR_KBD_NO_MORE_HANDLE
Ran out of handles.

441 ERROR_KBD_CANNOT_CREATE_KCB
Unable to create kcb.

442 ERROR_KBD_CODEPAGE_LOAD_INCOMPL
Unsuccessful code-page load.

443 ERROR_KBD_INVALID_CODEPAGE_ID
Code page identity is not valid.

444 ERROR_KBD_NO_CODEPAGE_SUPPORT
No code page support.

445 ERROR_KBD_FOCUS_REQUIRED
Keyboard focus required.

446 ERROR_KBD_FOCUS_ALREADY_ACTIVE
Calling thread has an outstanding focus.

447 ERROR_KBD_KEYBOARD_BUSY
Keyboard is busy.

448 ERROR_KBD_INVALID_CODEPAGE
Code page is not valid.

449 ERROR_KBD_UNABLE_TO_FOCUS
Focus attempt failed.

450 ERROR_SMG_SESSION_NON_SELECT
Session is not selectable.

451 ERROR_SMG_SESSION_NOT_FOREGRND
Parent/child session is not foreground.

452 ERROR_SMG_SESSION_NOT_PARENT
Not parent of requested child.

453 ERROR_SMG_INVALID_START_MODE
Session start mode is not valid.

454 ERROR_SMG_INVALID_RELATED_OPT
Session start related option is not valid.

455 ERROR_SMG_INVALID_BOND_OPTION
Session bond option is not valid.

456 ERROR_SMG_INVALID_SELECT_OPT
Session select option is not valid.

457 ERROR_SMG_START_IN_BACKGROUND
Session started in background.

458 ERROR_SMG_INVALID_STOP_OPTION
Session stop option is not valid.

459 ERROR_SMG_BAD_RESERVE
Reserved parameters are not zero.

460 ERROR_SMG_PROCESS_NOT_PARENT
Session parent process already exists.

461 ERROR_SMG_INVALID_DATA_LENGTH
Data length is not valid.

462 ERROR_SMG_NOT_BOUND
Parent is not bound.

463 ERROR_SMG_RETRY_SUB_ALLOC
Retry request block allocation.

464 ERROR_KBD_DETACHED
This call is not allowed for a detached PID.

465 ERROR_VIO_DETACHED
This call is not allowed for a detached PID.

466 ERROR_MOU_DETACHED
This call is not allowed for a detached PID.

467 ERROR_VIO_FONT
No font is available to support the mode.

468 ERROR_VIO_USER_FONT
User font is active.

469 ERROR_VIO_BAD_CP
Code page specified is not valid.

470 ERROR_VIO_NO_CP
System displays do not support code page.

471 ERROR_VIO_NA_CP
Current display does not support code page.

472 ERROR_INVALID_CODE_PAGE
Code page is not valid.

473 ERROR_CPLIST_TOO_SMALL
Code page list is too small.

474 ERROR_CP_NOT_MOVED
Code page was not moved.

475 ERROR_MODE_SWITCH_INIT
Mode switch initialization error.

476 ERROR_CODE_PAGE_NOT_FOUND
Code page was not found.

477 ERROR_UNEXPECTED_SLOT_RETURNED
Internal error.

478 ERROR_SMG_INVALID_TRACE_OPTION
Start session trace indicator is not valid.

479 ERROR_VIO_INTERNAL_RESOURCE
VIO internal resource error.

480 ERROR_VIO_SHELL_INIT
VIO shell initialization error.

481 ERROR_SMG_NO_HARD_ERRORS
No session manager hard errors.

482 ERROR_CP_SWITCH_INCOMPLETE
DosSetProcessCp is unable to set a KBD or VIO code page.

483 ERROR_VIO_TRANSPARENT_POPUP
Error during VIO pop-up window.

484 ERROR_CRITSEC_OVERFLOW
Critical section overflow.

485 ERROR_CRITSEC_UNDERFLOW
Critical section underflow.

486 ERROR_VIO_BAD_RESERVE
Reserved parameter is not zero.

487 ERROR_INVALID_ADDRESS
Physical address is not valid.

488 ERROR_ZERO_SELECTORS_REQUESTED
At least one selector must be requested.

489 ERROR_NOT_ENOUGH_SELECTORS_AVA
Not enough GDT selectors to satisfy request.

490 ERROR_INVALID_SELECTOR

GDT selector is not valid.

491 ERROR_SMG_INVALID_PROGRAM_TYPE
Program type is not valid.

492 ERROR_SMG_INVALID_PGM_CONTROL
Program control is not valid.

493 ERROR_SMG_INVALID_INHERIT_OPT
Inherit option is not valid.

494 ERROR_VIO_EXTENDED_SG

495 ERROR_VIO_NOT_PRES_MGR_SG

496 ERROR_VIO_SHIELD_OWNED

497 ERROR_VIO_NO_MORE_HANDLES

498 ERROR_VIO_SEE_ERROR_LOG

499 ERROR_VIO_ASSOCIATED_DC

Errors 500 to 599

The following shows the numerical value of an error, and its symbolic name.

500 ERROR_KBD_NO_CONSOLE

501 ERROR_MOUSE_NO_CONSOLE

502 ERROR_MOUSE_INVALID_HANDLE

503 ERROR_SMG_INVALID_DEBUG_PARMS

504 ERROR_KBD_EXTENDED_SG

505 ERROR_MOU_EXTENDED_SG

506 ERROR_SMG_INVALID_ICON_FILE

507 ERROR_TRC_PID_NON_EXISTENT

508 ERROR_TRC_COUNT_ACTIVE

509 ERROR_TRC_SUSPENDED_BY_COUNT

510 ERROR_TRC_COUNT_INACTIVE

511 ERROR_TRC_COUNT_REACHED

512 ERROR_NO_MC_TRACE

513 ERROR_MC_TRACE

514 ERROR_TRC_COUNT_ZERO

515 ERROR_SMG_TOO_MANY_DDS

516 ERROR_SMG_INVALID_NOTIFICATION

517 ERROR_LF_INVALID_FUNCTION

518 ERROR_LF_NOT_AVAIL

519 ERROR_LF_SUSPENDED

520 ERROR_LF_BUF_TOO_SMALL

521 ERROR_LF_BUFFER_CORRUPTED

521 ERROR_LF_BUFFER_FULL

522 ERROR_LF_INVALID_DAEMON

522 ERROR_LF_INVALID_RECORD

523 ERROR_LF_INVALID_TEMPL

523 ERROR_LF_INVALID_SERVICE

524 ERROR_LF_GENERAL_FAILURE

525 ERROR_LF_INVALID_ID

526 ERROR_LF_INVALID_HANDLE

527 ERROR_LF_NO_ID_AVAIL

528 ERROR_LF_TEMPLATE_AREA_FULL

529 ERROR_LF_ID_IN_USE

530 ERROR_MOU_NOT_INITIALIZED

531 ERROR_MOUINITREAL_DONE

532 ERROR_DOSSUB_CORRUPTED

533 ERROR_MOUSE_CALLER_NOT_SUBSYS

534 ERROR_ARITHMETIC_OVERFLOW

535 ERROR_TMR_NO_DEVICE

536 ERROR_TMR_INVALID_TIME

537 ERROR_PVW_INVALID_ENTITY

538 ERROR_PVW_INVALID_ENTITY_TYPE

539 ERROR_PVW_INVALID_SPEC

540 ERROR_PVW_INVALID_RANGE_TYPE

541 ERROR_PVW_INVALID_COUNTER_BLK

542 ERROR_PVW_INVALID_TEXT_BLK

543 ERROR_PRF_NOT_INITIALIZED

544 ERROR_PRF_ALREADY_INITIALIZED

545 ERROR_PRF_NOT_STARTED

546 ERROR_PRF_ALREADY_STARTED

547 ERROR_PRF_TIMER_OUT_OF_RANGE

548 ERROR_PRF_TIMER_RESET

549-599 Reserved.

Errors 600 to 1999

The following shows the numerical value of an error, and its symbolic name.

600-638 Reserved.

639 ERROR_VDD_LOCK_USEAGE_DENIED

640 ERROR_TIMEOUT

641 ERROR_VDM_DOWN

642 ERROR_VDM_LIMIT

643 ERROR_VDD_NOT_FOUND

644 ERROR_INVALID_CALLER

645 ERROR_PID_MISMATCH

646 ERROR_INVALID_VDD_HANDLE

647 ERROR_VLPT_NO_SPOOLER

648 ERROR_VCOM_DEVICE_BUSY

649 ERROR_VLPT_DEVICE_BUSY

650 ERROR_NESTING_TOO_DEEP

651 ERROR_VDD_MISSING

691 ERROR_IMP_INVALID_PARM

692 ERROR_IMP_INVALID_LENGTH

693 MSG_HPFS_DISK_ERROR_WARN

730 ERROR_MON_BAD_BUFFER

731 ERROR_MODULE_CORRUPTED

732-1476 Reserved.

1477 ERROR_SM_OUTOF_SWAPFILE

1478-1999 Reserved.

Errors 2000 to 59999

The following shows the numerical value of an error, and its symbolic name.

2000-2054 Reserved.

2055 ERROR_LF_TIMEOUT

2057 ERROR_LF_SUSPEND_SUCCESS

2058 ERROR_LF_RESUME_SUCCESS

2059 ERROR_LF_REDIRECT_SUCCESS

2060 ERROR_LF_REDIRECT_FAILURE

32768 ERROR_SWAPPER_NOT_ACTIVE

32769 ERROR_INVALID_SWAPID

32770 ERROR_IOERR_SWAP_FILE

32771 ERROR_SWAP_TABLE_FULL

32772 ERROR_SWAP_FILE_FULL

32773 ERROR_CANT_INIT_SWAPPER

32774 ERROR_SWAPPER_ALREADY_INIT

32775 ERROR_PMM_INSUFFICIENT_MEMORY

32776 ERROR_PMM_INVALID_FLAGS

32777 ERROR_PMM_INVALID_ADDRESS

32778 ERROR_PMM_LOCK_FAILED

32779 ERROR_PMM_UNLOCK_FAILED

32780 ERROR_PMM_MOVE_INCOMPLETE

32781 ERROR_UCOM_DRIVE_RENAMED

32782 ERROR_UCOM_FILENAME_TRUNCATED

32783 ERROR_UCOM_BUFFER_LENGTH

32784 ERROR_MON_CHAIN_HANDLE

32785 ERROR_MON_NOT_REGISTERED

32786 ERROR_SMG_ALREADY_TOP

32787 ERROR_PMM_ARENA_MODIFIED

32788 ERROR_SMG_PRINTER_OPEN

32789 ERROR_PMM_SET_FLAGS_FAILED

32790 ERROR_INVALID_DOS_DD

32791 ERROR_BLOCKED

32792 ERROR_NOBLOCK

32793 ERROR_INSTANCE_SHARED

32794 ERROR_NO_OBJECT

32795 ERROR_PARTIAL_ATTACH

32796 ERROR_INCACHE

32797 ERROR_SWAP_IO_PROBLEMS

32798 ERROR_CROSSES_OBJECT_BOUNDARY

32799 ERROR_LONGLOCK

32800 ERROR_SHORTLOCK

32801 ERROR_UVIRTLOCK

32802 ERROR_ALIASLOCK

32803 ERROR_ALIAS

32804 ERROR_NO_MORE_HANDLES

32805 ERROR_SCAN_TERMINATED

32806 ERROR_TERMINATOR_NOT_FOUND

32807 ERROR_NOT_DIRECT_CHILD

32808 ERROR_DELAY_FREE

32809 ERROR_GUARDPAGE

32900 ERROR_SWAPERROR

32901 ERROR_LDRERROR

32902 ERROR_NOMEMORY

32903 ERROR_NOACCESS

32904 ERROR_NO_DLL_TERM

32905-59999 Reserved.

Errors 60000 to 65079

The following shows the numerical value of an error, and its symbolic name.

60000-65025 Reserved.

65026 ERROR_CPSIO_CODE_PAGE_INVALID

65027 ERROR_CPSIO_NO_SPOOLER

65028 ERROR_CPSIO_FONT_ID_INVALID

65029-65032 Reserved.

65033 ERROR_CPSIO_INTERNAL_ERROR

65034 ERROR_CPSIO_INVALID_PTR_NAME

65035-65036 Reserved.

65037 ERROR_CPSIO_NOT_ACTIVE

65038 Reserved.

65039 ERROR_CPSIO_PID_FULL

65040 ERROR_CPSIO_PID_NOT_FOUND

65041-65042 Reserved.

65043 ERROR_CPSIO_READ_CTL_SEQ

65044 Reserved.

65045 ERROR_CPSIO_READ_FNT_DEF

65046 Reserved.

65047 ERROR_CPSIO_WRITE_ERROR

65048 ERROR_CPSIO_WRITE_FULL_ERROR

65049 ERROR_CPSIO_WRITE_HANDLE_BAD

65050-65073 Reserved.

65074 ERROR_CPSIO_SWIT_LOAD

65075-65076 Reserved.

65077 ERROR_CPSIO_INV_COMMAND

65078 ERROR_CPSIO_NO_FONT_SWIT

65079 ERROR_ENTRY_IS_CALLGATE

Debugging

Debugging is the process of detecting, diagnosing, and eliminating errors in programs. A debugger application is designed to interact with and control the application that it is debugging. Because of the protected mode architecture of OS/2, special steps must be taken to enable a debugger application to perform its functions in the application being debugged (for example, to examine and manipulate memory locations in the address space of another process).

The following topic is related to the information in this chapter:

- Program execution and control

About Debugging

DosDebug enables one application to control the execution of another application for debugging purposes.

An application is selected for debugging when it is started. DosExecPgm and DosStartSession both have flags that can be used to specify that the application being started is to be controlled by the starting application.

DosExecPgm starts an application within a new process. DosStartSession starts a new session within which one or more processes can be executing. See DosStartSession and DosExecPgm for details on how to start an application for debugging purposes. For information on processes and sessions, see Program Execution Control in this book.

Once a process has been selected for debugging, DosDebug is used to control its execution and to examine and manipulate its variables.

DosDebug provides a full set of debugging commands, including execution control commands-like single stepping and setting watchpoints-and commands to examine and manipulate the memory and registers of the process being debugged. The debugger process can access specific threads within a process being debugged and specific processes within a session being debugged.

DosDebug also has a rich set of notification messages to keep the debugger application informed of activities occurring during the execution of the application being debugged.

The debugger application can use the session and process control functions described in Program Execution Control to control the child process or session being debugged. For example, the debugger can use DosSelectSession to switch itself, or the session being debugged, to the foreground.

Using the Debugging Function

DosDebug provides a set of commands that permit one process to control another process for debugging.

In the following code fragment, the calling process uses DosDebug to modify a word in a controlled process. All the necessary steps have already been taken so that the calling process controls the second process-the process identifier of the controlled process has been placed into *PID* , the address of the word to be modified in the controlled process has been placed into *Addr* , and the value to be substituted in the controlled process has been placed into *Value* .

(Due to the size of the *debug_buffer* data structure, the code fragment has been divided into two figures. If you were actually entering this into a program, the information would be together as if it were all one figure.)

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

```
#define INCL_DOSPROCESS   /* Process and thread values */
#include <os2.h>
#include <stdio.h>
```

```
struct debug_buffer {
    ULONG   Pid;        /* Debuggee Process ID       */
    ULONG   Tid;        /* Debuggee Thread ID        */
    LONG    Cmd;        /* Command or Notification   */
    LONG    Value;      /* Generic Data Value        */
    ULONG   Addr;       /* Debuggee Address          */
    ULONG   Buffer;     /* Debugger Buffer Address   */
    ULONG   Len;        /* Length of Range           */
    ULONG   Index;      /* Generic Identifier Index  */
    ULONG   MTE;        /* Module Table Entry Handle */
    ULONG   EAX;        /* Register Set              */
    ULONG   ECX;
    ULONG   EDX;
    ULONG   EBX;
    ULONG   ESP;
    ULONG   EBP;
    ULONG   ESI;
    ULONG   EDI;
    ULONG   EFlags;
    ULONG   EIP;
    ULONG   CSLim;      /* Byte Granular Limits      */
    ULONG   CSBase;     /* Byte Granular Base        */
    UCHAR   CSAcc;      /* Access Bytes              */
    UCHAR   CSAtr;      /* Attribute Bytes           */
    USHORT  CS;
    ULONG   DSLim;
    ULONG   DSBase;
    UCHAR   DSAcc;
    UCHAR   DSAtr;
    USHORT  DS;
    ULONG   ESLim;
    ULONG   ESBase;
    UCHAR   ESAcc;
    UCHAR   ESAtr;
    USHORT  ES;
    ULONG   FSLim;
    ULONG   FSBase;
    UCHAR   FSAcc;
    UCHAR   FSAtr;
    USHORT  FS;
    ULONG   GSLim;
    ULONG   GSBase;
    UCHAR   GSAcc;
    UCHAR   GSAtr;
    USHORT  GS;
    ULONG   SSLim;
    ULONG   SSBase;
    UCHAR   SSAcc;
    UCHAR   SSAtr;
    USHORT  SS;
};
```

```
struct  debug_buffer  DbgBuf;      /* Debug buffer                          */
ULONG   ulPID;                     /* Process ID of the controlled process */
ULONG   ulAddr;                    /* Address in the controlled process    */
LONG    lValue;                    /* Value to be substituted in the       */
                                   /* controlled process                   */
APIRET  ulrc;                      /* Return code                          */

DbgBuf.Cmd = DBG_C_WriteMem;       /* Indicate that a Write Word           */
                                   /* command is requested                 */

DbgBuf.Pid = ulPID;                /* Place PID of controlled process      */
                                   /* into the debug buffer                */

DbgBuf.Addr = ulAddr;              /* Place the word address (within the   */
                                   /* controlled process) into the debug   */
                                   /* buffer                               */

DbgBuf.Value = lValue;             /* Place the value to be updated into   */
                                   /* the specified word of the controlled */
                                   /* process                              */

ulrc = DosDebug(&DbgBuf);

if (ulrc != 0) {
   printf("DosDebug error: return code = %ld",
          ulrc);
```

```
    return;
}

/* Be sure to check DbgBuf.Cmd for the notification returned by DosDebug */
```

The *Cmd* field in the debug buffer is used for two purposes. On input, the *Cmd* field is used to pass the commands that direct DosDebug's activities. On output, the *Cmd* field is used by DosDebug to return a notification indicating the events and activities that occurred during the call.

If DosDebug returns no error, a notification resides in the *Cmd* field of the debug buffer. The data returned with the notification varies, depending on the command passed in the *Cmd* field of the debug buffer data structure when DosDebug was called.

Not all fields in the debug buffer have to be defined on every DosDebug command. The same field can have a different meaning in different DosDebug commands.

Some notifications (such as DBG_N_ModuleLoad and DBG_N_NewProc) might require multiple returns to the debugger. These additional, pending notifications will be returned before the process being debugged is permitted to execute any more user code, and will be returned on the Go, Single Step, or Stop commands.

Additional notifications can be pending at any time, so a debugger must be ready to handle any notification any time a Go, Single Step, or Stop command is called.

Debugging on OS/2 Warp (PowerPC Edition)

The following are the specific features available for debugging for OS/2 Warp (PowerPC Edition):

1. **PowerPC-specific debug buffer:** The debug buffer is specified in the value field of DBG_C_Connect command as follows:
 - Intel:

     ```
     DBG_L_386
     ```

 This is not defined in bsedos.h.
 - PowerPC:

     ```
     DBG_L_PPC
     ```

 This is defined in bsedos.h.
2. **PowerPC-specific functions:**
 a. **DBG_C_ATTACH:** This debug command has the following parameters:

PID	Process ID of debuggee
CMD	DBG_C_Attach
VALUE	DBG_L_PPC

 This command is defined as follows:
 - For clients in bsedos.h
 - For servers in server\include\debug_types.h as follows:

     ```
     #define DBG_L_PPC       2
     ```

 This command returns:

DBG_N_Success	Attachment made
DBG_N_Error	Any OS/2-defined error

 This command allows attaching to a currently running task.

 Note: A DBG_C_Connect does not need to be issued because DBG_C_Attach will perform the connection. Also, because the debugger did not start the task, it will not have a parent/child relationship as in a DBG_C_Connect.

DosDebug generates the following notifications:

1. DBG_N_ModuleLoad notifications for all loaded modules
2. DBG_N_ThreadCreate notifications for all active threads in task

Note: Unlike DBG_C_Connect, there will not be any DBG_N_ModuleInit notifications because the task is most likely already in _main.

b. **DBG_C_Detach:** This debug command has the following parameters:

PID	Process ID of debuggee
CMD	DBG_C_Detach

This command returns:

DBG_N_Success	Attachment made
DBG_N_Error	Any OS/2-defined error

This command detaches from attached and connected tasks. The specified task is resumed and all debugging hooks are removed.

Note: This is the only call that will cleanly turn off debugging and resume the specified task. DBG_C_Term will kill the task whether it was connected or attached.

3. **PowerPC-specific notifications:** are as follows:

a. DBG_N_ModuleInit: Module initialization routine about to run. This notification is the same format as DBG_N_ModuleLoad.

CMD	DBG_N_ModuleInit
Value	MTE (module handle)
Addr	0

b. DBG_N_ReadyToRunMain: Debuggee thread 1 ready to run _start. Issued after all import DLL Init/Term routines have completed. This allows the debugger to know that the debugged task has finished loading and is ready to run.

Note: For descendant debuggee tasks, DBG_N_NewProc will be sent instead.

CMD	DBG_N_ReadyToRunMain
Value	Process ID of debuggee

4. **DBG_C_Go and DBG_C_SStep:** will stop and return any pending notifications.

5. **Serialization of command execution:** All notifications other than DBG_N_Success and DBG_N_Error must be acknowledged through DBG_C_Continue before processing resumes or another notification can be grabbed. Thus, only GO/SStep and Stop (to prevent retrieving a pending notification) will be prevented from running. All other subcommands can be executed.

DosDebug informs the debugger that a notification has been returned, but not acknowledged. If you issue a Go/SStep/Stop and get the following synchronous notification, then you must issue a DBG_C_Continue before resuming execution.

```
rc = DosDebug() = 0;
puDB->Cmd   = DBG_N_Error;
puDB->Value = ERROR_INVALID_FUNCTION;
```

The following code segments illustrate the use of various commands:

```
/*********************************************************************/
/* Connect and grab all pending DBG_N_ThreadCreate, DBG_N_ModuleInit */
/* and DBG_N_ModuleLoad notifications. Note, Stop returns            */
/* DBG_N_Success when there are no more notifications.               */
/*********************************************************************/

      DBG_C_Connect

      DBG_C_Stop
      while ( puDB->Cmd != DBG_N_Success ) {
        DBG_C_Continue w XCPT_CONTINUE_STOP
        DBG_C_Stop
      }
```

```
/*********************************************************************/
/* Go until certain notification occurs. This loop will also grab any */
/* pending notifications that are outstanding since DBG_C_Go won't    */
/* execute if any notifications are pending.                          */
/*********************************************************************/


      DBG_C_Go
      while (debug_buffer->Cmd != Notification your looking for) {

         DBG_C_Continue w XCPT_CONTINUE_STOP
         DBG_C_GO
      }




/*********************************************************************/
/* Clear all pending notifications and acknowledge each notification */
/*********************************************************************/

      DBG_C_STOP
      while (debug_buffer->Cmd != DBG_N_Success) {
         DBG_C_Continue w XCPT_CONTINUE_STOP
         DBG_C_STOP
      }
```

6. **Per task serialization of all DosDebug subcommands:** A debugger can be multi-threaded. If the debugger issues subcommands for different tasks they can run concurrently. If the debugger issues subcommands for the same task they will be serialized.

7. **Breakpoint exceptions (trap word instruction):** Debugger must increment IP to next instruction (+4) whenever a breakpoint exception occurs.

8. **DBG_N_Exception notification:** Returns Thread ID of thread taking exception in debug_buffer->TID field. When responding to exception notifications use the proper Thread ID in the DBG_C_Continue call.

9. **Unsupported or not working for the PowerPC:**

 a. Calls to these commands will result in a DBG_C_Null:
 - DBG_C_XchngOpcode
 - DBG_C_RangeStep
 - DBG_C_MapRWAlias
 - DBG_C_UnMapAlias
 - DBG_C_MapROAlias
 - DBG_C_LinToSel
 - DBG_C_SelToLin

 b. Global scope watchpoints: Watchpoint effective in the context of any task currently not allowed.

 c. Descendant (task or session) debugging not working.
 1. EXEC_ASYNCRESULTDB: Tasking flag will act like EXEC_TRACE
 2. SSF_TRACEOPT_TRACEALL: Session flag will act like SSF_TRACEOPT_TRACE
 3. DBG_N_ProcNew notification never sent

10. **DBG_C_NumToAddr and DBG_AddrToNum:** these functions are not fully supported on the PowerPC as there meaning differs between Intel and PowerPC architectures. The following has been observed:

 a. On Intel, this should be data segment. It appears to be data segment on the PowerPC:

```
PBuf->Cmd = DBG_C_NumToAddr;
PBuf->Value = 1;
```

 b. On Intel, this should be a code segment. On the PowerPC, this is invalid:

```
PBuf->Cmd = DBG_C_NumToAddr;
PBuf->Value = 2;
```

 c. On Intel, this should be invalid. On the PowerPC, this appears to be code segment:

```
PBuf->Cmd = DBG_C_NumToAddr;
PBuf->Value = 0;
```

Summary of bsedos.h

The following summarizes the contents of the header file bsedos.h.

Debug Buffer for the PowerPC

```
typedef struc uDB {
  ulong Pid;            /* Debuggee Process id          */
  ulong Tid;            /* Debuggee Thread id           */
  long  Cmd;            /* Command or Notification      */
  long  Value;          /* Generic Data Value           */
  ulong Addr;           /* Debuggee Address             */
  ulong Buffer;         /* Debugger Buffer Address      */
  ulong Len;            /* Length of Range              */
  ulong Index;          /* Generic Identifier Index     */
  ulong MTE;            /* Module Table Entry Handle    */
  ulong ctr;            /* Count register               */
  ulong lr;             /* Link register                */
  ulong xer;            /* Integer Exception register   */
  ulong msr;            /* Machine State Register       */
  ulong cr;             /* Condition Register           */
  ulong iar;            /* Instruction pointer          */
  ulong GP_REGS[32];    /* General Purpose registers    */
} uDB_t
```

DosDebug Command Numbers

The following are the DosDebug command numbers:

```
#define DBG_C_Null              0       /* Null                         */
#define DBG_C_ReadMem           1       /* Read Word                    */
#define DBG_C_ReadMem_I         1       /* Read Word                    */
#define DBG_C_ReadMem_D         2       /* Read Word (same as 1)        */
#define DBG_C_ReadReg           3       /* Read Register Set            */
#define DBG_C_WriteMem          4       /* Write Word                   */
#define DBG_C_WriteMem_I        4       /* Write Word                   */
#define DBG_C_WriteMem_D        5       /* Write Word (same as 4)       */
#define DBG_C_WriteReg          6       /* Write Register Set           */
#define DBG_C_Go                7       /* Go                           */
#define DBG_C_Term              8       /* Terminate                    */
#define DBG_C_SStep             9       /* Single Step                  */
#define DBG_C_Stop              10      /* Stop                         */
#define DBG_C_Freeze            11      /* Freeze Thread                */
#define DBG_C_Resume            12      /* Resume Thread                */
#define DBG_C_NumToAddr         13      /* Object Number to Address     */
#define DBG_C_ReadCoRegs        14      /* Read Coprocessor Registers   */
#define DBG_C_WriteCoRegs       15      /* Write Coprocessor Registers  */
                                        /* 16 is reserved               */
#define DBG_C_ThrdStat          17      /* Get Thread Status            */
#define DBG_C_Connect           21      /* Connect to Debuggee          */
#define DBG_C_ReadMemBuf        22      /* Read Memory Buffer           */
#define DBG_C_WriteMemBuf       23      /* Write Memory Buffer          */
```

```
#define DBG_C_SetWatch            24      /* Set Watchpoint               */
#define DBG_C_ClearWatch          25      /* Clear Watchpoint             */
#define DBG_C_Continue            27      /* Continue after an Exception  */
#define DBG_C_AddrToObject        28      /* Address to Object            */

#define DBG_C_Attach              32      /* Attach to task               */
#define DBG_C_Detach              33      /* Detach task                  */
#define DBG_C_SetBreak            34      /* Set breakpoint for PPC       */
#define DBG_C_ClrBreak            35      /* Clear breakpoint for PPC     */
```

The following defines are not implemented for the PowerPC and are translated to DBG_C_NULL:

```
#define DBG_C_MapROAlias        18      /* Map read-only alias          */
#define DBG_C_MapRWAlias        19      /* Map read-write alias         */
#define DBG_C_UnMapAlias        20      /* Unmap Alias                  */
#define DBG_C_RangeStep         26      /* Range Step                   */
#define DBG_C_XchgOpcode        29      /* Exchange opcode and go       */
#define DBG_C_LinToSel          30      /* 32 to 16 conversion          */
#define DBG_C_SelToLin          31      /* 16 to 32 conversion          */
```

The following defines are not yet implemented for the PowerPC:

```
#define DBG_C_SetBreak         34      /* Set breakpoint for PPC       */
#define DBG_C_ClrBreak         35      /* Clear breakpoint for PPC     */
```

DosDebug Notification Numbers

The following are the DosDebug notification numbers for the PowerPC:

```
#define DBG_N_Success            0L     /* Command completed successfully  */
#define DBG_N_Error             -1L     /* Error detected during command   */
#define DBG_N_ProcTerm          -6L     /* Process exiting - ExitList done */
#define DBG_N_Exception         -7L     /* Exception detected              */
#define DBG_N_ModuleLoad        -8L     /* Module loaded                   */
#define DBG_N_CoError           -9L     /* Coprocessor not in use error    */
#define DBG_N_ThreadTerm        -10L    /* Thread exiting - Exitlist soon  */
#define DBG_N_AsyncStop         -11L    /* Async Stop detected             */
#define DBG_N_NewProc           -12L    /* New Process started             */
#define DBG_N_AliasFree         -13L    /* Alias needs to be freed         */
#define DBG_N_Watchpoint        -14L    /* Watchpoint hit                  */
#define DBG_N_ThreadCreate      -15L    /* New thread created              */
#define DBG_N_ModuleFree        -16L    /* Module freed                    */
#define DBG_N_RangeStep         -17L    /* Range Step completed            */

/*** NEW to PowerPC ***/
#define DBG_N_ModuleInit        -18L    /* Module init routine about to run*/
#define DBG_N_ReadyToRunMain    -19L    /* EXE ready to run, all import DLL
                                           Init routines completed */

#define DBG_L_PPC       2
```

DBG_T_ TState Values

These are the possible values that can be returned in the TState field of the TStat structure. These values identify scheduler state information:

```
#define DBG_T_Runnable  0
#define DBG_T_Suspended 1
#define DBG_T_Blocked   2
#define DBG_T_CritSec   3
```

DosDebug Hardware-Specific Subcommands

The following are the DosDebug PowerPC-specific subcommands:

1. Connect:

   ```
   uDB.Value = DBG_L_PPC /* Debugging level number */
   ```

 DBG_L_PPC is defined in bsedos.h.

2. ReadRegs and WriteRegs:

 Uses the debug buffer defined in bsedos.h.

3. ReadCoRegs and WriteCoRegs:

 PPC_FLOAT_STATE_SIZE is defined in public uKernel release tree in file thread_status.h:

   ```
   uDB.Value = DBG_CO_PPC  /* Coprocessor type identifier       */
   uDB.Len   = PPC_FLOAT_STATE_SIZE /* Size of coprocessor     */
                                    /* register context buffer */
   ```

   ```
   #define DBG_CO_PPC 2
   ```

 Note: Not defined in bsedos.h.

4. ThrdStat. uDB.Len is the length of the thread status buffer, in bytes. This field is four bytes (same as Intel).

 Thread Status Buffer is the same as Intel and is not stored in bsedos.h.

   ```
   TStat struc
      unsigned char DbgState  : DBG_D_Thawed, DBG_D_Frozen
      unsigned char TState    : DBG_T_Runnable, _Suspended, _Blocked, _CritSec
      unsigned char TPriority
   TStat ends
   ```

 The following are defined in this *Guide* :

   ```
   #define DBG_D_Thawed    0
   #define DBG_D_Frozen    1
   ```

 The following are defined in bsedos.h and in this *Guide* :

   ```
   #define DBG_T_Runnable  0
   #define DBG_T_Suspended 1
   #define DBG_T_Blocked   2
   #define DBG_T_CritSec   3
   ```

5. SetWatch:

 uDB.Addr Starting Address of Watchpoint

uDB.Len	Length of Watchpoint, in bytes	
uDB.Index	Reserved (0)	
uDB.Value	Watchpoint Type and Scope (same as Intel)	
	Scopes	Only DBG_W_Local allowed. No DBG_W_Global allowed.
	Types	DBG_W_Execute, DBG_W_Write, and DBG_W_ReadWrite.

All defines are the same as the Intel defines, as described in this *Guide* .

Note: There are two types of Watchpoints:

a. Instruction (DBG_W_Execute). Four bytes long and four byte aligned.

b. Data (DBG_W_Write, DBG_W_ReadWrite). Eight bytes long and eight byte aligned.

```
601, 604 : 1 instruction WP, 1 data WP
603      : 1 instruction WP
```

6. Attach and Detach:

DBG_C_ATTACH - Debug command:

Parameters:

PID	Process ID of debuggee
CMD	DBG_C_Attach
VALUE	DBG_L_PPC (DBG_L_PPC - defined in bsedos.h)

Returns:

Note: A DBG_C_Connect does not need to be issued because DBG_C_Attach will perform the connection. Also, because the debugger did not start the task, it will not have a parent/child relationship as in a DBG_C_Connect.

DBG_N_Success	Attachment made.
DBG_N_Error	All errors (for example, bad process id)

DosDebug will generate the following notifications:

a. DBG_N_ModuleLoad notifications for all loaded modules.
b. DBG_N_ThreadCreate notifications for all active threads in task.

Currently, descendant debugging is not working.

DBG_C_Detach - Debug command.

Parameters:

PID	Process ID of debuggee
CMD	DBG_C_Detach

Returns:

DBG_N_Success	Attachment made
DBG_N_Error	All errors (for example, bad process id)

Will detach from an attached or connected task.

Note: This is the only call that will normally turn off debugging and resume the specified task. DBG_C_Term will kill the task whether it was connected or attached.

DosDebug Commands

The following table list describes the available commands.

Cmd No.	Command Name	Description
0	DBG_C_Null	Null
1	DBG_C_ReadMem	Read Word
1	DBG_C_ReadMem_I	Read Word
2	DBG_C_ReadMem_D	Read Word (same as 1)
3	DBG_C_ReadReg	Read Register Set
4	DBG_C_WriteMem	Write Word
4	DBG_C_WriteMem_I	Write Word
5	DBG_C_WriteMem_D	Write Word (same as 4)
6	DBG_C_WriteReg	Write Register Set
7	DBG_C_Go	Go
8	DBG_C_Term	Terminate
9	DBG_C_SStep	Single Step
10	DBG_C_Stop	Stop
11	DBG_C_Freeze	Freeze Thread
12	DBG_C_Resume	Resume Thread
13	DBG_C_NumToAddr	Object Number to Address
14	DBG_C_ReadCoRegs	Read Coprocessor Registers
15	DBG_C_WriteCoRegs	Write Coprocessor Registers
16	Reserved	Reserved
17	DBG_C_ThrdStat	Get Thread Status
18	DBG_C_MapROAlias	Map Read-Only Alias
19	DBG_C_MapRWAlias	Map Read-Write Alias
20	DBG_C_UnMapAlias	Unmap Alias
21	DBG_C_Connect	Connect to Debuggee
22	DBG_C_ReadMemBuf	Read Memory Buffer
23	DBG_C_WriteMemBuf	Write Memory Buffer
24	DBG_C_SetWatch	Set Watchpoint
25	DBG_C_ClearWatch	Clear Watchpoint
26	DBG_C_RangeStep	Range Step
27	DBG_C_Continue	Continue After an Execution
28	DBG_C_AddrToObject	Address to an Object
29	DBG_C_XchngOpcode	Exchange Opcode and Go
30	DBG_C_LinToSel	Translate Linear Address to Segment:Offset
31	DBG_C_SelToLin	Translate Segment:Offset to Linear Address

Not all fields must be defined for every DosDebug command. The same field can have a different meaning in different DosDebug commands. Fields in the DosDebug Buffer structure that are not listed in the *Parameter* section of each command is not useful for that command.

Error cases for commands are not listed. The listed return values from commands are valid only if the DBG_N_Success notification is given.

DBG_C_Null

Debug Command 0 - Null Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_Null

No operation is performed on the debuggee. You can issue this command to verify the process ID of the debuggee, and to check if the debuggee is active.

Pid must be valid, or an error is returned.

DBG_C_ReadMem

Debug Command 1 and 2 - Read Word Command

Parameters

Pid = Process ID of debuggee
Addr = Address to read from
Cmd = DBG_C_ReadMem_I, or DBG_C_ReadMem_D, or DBG_C_ReadMem

The commands DBG_C_ReadMem_I, DBG_C_ReadMem_D, and DBG_C_ReadMem are identical.

Returns

The word at the desired address is read, and stored into Value.

Value = Word read from the specified address.

Restrictions

You are unable to read from any memory outside user space.

The high-order word of Value is set to zero.

DBG_C_ReadMem_I

Debug Command 1 and 2 - Read Word Command

Parameters

Pid = Process ID of debuggee
Addr = Address to read from
Cmd = DBG_C_ReadMem_I, or DBG_C_ReadMem_D, or DBG_C_ReadMem

The commands DBG_C_ReadMem_I, DBG_C_ReadMem_D, and DBG_C_ReadMem are identical.

Returns

The word at the desired address is read, and stored into Value.

Value = Word read from the specified address.

Restrictions

You are unable to read from any memory outside user space.

The high-order word of Value is set to zero.

DBG_C_ReadMem_D

Debug Command 1 and 2 - Read Word Command

Parameters

Pid = Process ID of debuggee
Addr = Address to read from
Cmd = DBG_C_ReadMem_I, or DBG_C_ReadMem_D, or DBG_C_ReadMem

The commands DBG_C_ReadMem_I, DBG_C_ReadMem_D, and DBG_C_ReadMem, are identical.

Returns

The word at the desired address is read, and stored into Value.

Value = Word read from the specified address.

Restrictions

You are unable to read from any memory outside user space.

The high-order word of Value is set to zero.

DBG_C_ReadReg

Debug Command 3 - Read Register Set Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of register set to read
Cmd = DBG_C_ReadReg

If Tid is zero and the debuggee is stopped, the register set comes from the active debuggee thread. If Tid is zero and the debuggee is executing, ERROR_INVALID_THREADID is returned.

Returns

The register set in the DosDebug Buffer structure is updated, including the selector information as follows:

Tid = Thread ID corresponding to the register set
MTE = Program module's MTE (Module Table Entry) handle

DBG_C_WriteMem

Debug Command 4 and 5 - Write Word Command

Parameters

Pid = Process ID of debuggee
Addr = Address to write to
Value = Word to write
Cmd = DBG_C_WriteMem_I, or DBG_C_WriteMem_D or DBG_C_WriteMem

The commands DBG_C_WriteMem_I, DBG_C_WriteMem_D, and DBG_C_WriteMem are identical.

Returns

The word in Value is written to the specified address.

In the case of a write to shared read-only memory, the memory is converted to private, and any set dynamic RAS tracepoints are removed from that memory, before the write is performed.

The area will continue to be shared by other processes, if any. In this way, breakpoints may be set in the debuggee without affecting the other modules.

Restrictions

You are unable to write to any memory outside user space.

The high-order word of Value is ignored.

DBG_C_WriteMem_I

Debug Command 4 and 5 - Write Word Command

Parameters

Pid = Process ID of debuggee
Addr = Address to write to
Value = Word to write
Cmd = DBG_C_WriteMem_I, or DBG_C_WriteMem_D, or DBG_C_WriteMem_I

The commands DBG_C_WriteMem_I, DBG_C_WriteMem_D, and DBG_C_WriteMem_I are identical.

Returns

The word in Value is written to the specified address.

In the case of a write to shared read-only memory, the memory is converted to private, and any set dynamic RAS tracepoints are removed from that memory, before the write is performed.

The area will continue to be shared by other processes, if any. In this way, breakpoints may be set in the debuggee without affecting the other modules.

Restrictions

You are unable to write to any memory outside user space.

The high-order word of Value is ignored.

DBG_C_WriteMem_D

Debug Command 4 and 5 - Write Word Command

Parameters

Pid = Process ID of debuggee
Addr = Address to write to
Value = Word to write
Cmd = DBG_C_WriteMem_I or DBG_C_WriteMem_D, or DBG_C_WriteMem, and

The commands DBG_C_WriteMem_I, DBG_C_WriteMem_D, and DBG_C_WriteMem, and are identical.

Returns

The word in Value is written to the specified address.

In the case of a write to shared read-only memory, the memory is converted to private, and any set dynamic RAS tracepoints are removed from that memory, before the write is performed.

The area will continue to be shared by other processes, if any. In this way, breakpoints may be set in the debuggee without affecting the other modules.

Restrictions

You are unable to write to any memory outside user space.

The high-order word of Value is ignored.

DBG_C_WriteReg

Debug Command 6 - Write Register Set Command

Parameters

Pid = Process ID of debuggee
Tid = Nonzero Thread ID of register set to write
Cmd = DBG_C_WriteReg

The register set in the DosDebug Buffer should contain the desired values.

Returns

Tid Thread ID corresponding to the register set.

All registers are updated to the stored values. The access rights, limits, and Eflags are also updated to match the current values.

An error is returned if the selectors are not accessible by user space code, or are not valid memory segments.

Restrictions

Reserved system or processor flags bits are not modified via this method, but are masked to their correct values. The selector access rights and limits cannot be modified. The flags, access rights, and limits in the DosDebug Buffer structure are updated to the actual values.

DBG_C_Go

Debug Command 7 - Go Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_Go

Returns

All non-frozen threads of the debuggee are allowed to execute user code at once. If all of the debuggee threads are frozen, an error is returned.

The DBG_C_Go command completes when a DosDebug event (such as a Breakpoint) occurs. This event can be any one of the DosDebug notifications. See DosDebug Notifications for more information.

When the next DosDebug event occurs, all threads in the debuggee process are marked to not execute any additional user code until the next DBG_C_Go command is issued. This provides a stable environment for debugging.

When the DBG_C_Go command returns, the register set is automatically updated to reflect the thread that detected the event.

DBG_C_Term

Debug Command 8 - Terminate Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_Term

Returns

The debuggee process is terminated immediately.

No additional DosDebug commands or notifications will be allowed to this process. Outstanding memory aliases and watchpoints will be invalidated automatically.

Debuggee DosExecPgm processing will be attempted, but any unexpected DosDebug event (such as a Breakpoint) during this period will cause the process to terminate without completing DosExitList processing. For this reason, data watchpoints will automatically be cleared before attempting DosExitList processing.

If the Terminate command is issued during DosExitList processing, DosExitList processing will terminate immediately, without continuing the DosExitList routines.

DBG_C_SStep

Debug Command 9 - Single Step Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of thread to single-step
Cmd = DBG_C_SStep

If Tid is zero, all threads will be marked to single-step at once, and the first thread to be scheduled to execute user-space code will single-step. No other threads will single-step.

Returns

Usually, the DBG_N_Exception notification is returned, but any notification may be returned. See DosDebug Notifications for more information.

Callgates that result in a privilege level transition to ring 0 will appear to single-step as a single instruction, with the single-step occurring just after the function completes. This hides ring 0 execution from debuggers.

Attempting to single-step any thread that is frozen results in an error.

Restrictions

The DBG_C_SStep command has two modes of operation, as follows:

- If Tid is zero, the current thread is single-stepped while allowing all other threads to execute.
- If Tid is nonzero, a specific thread is selected for single-stepping. Only that thread is executed, even if it is single-stepping a kernel function that can potentially cause a deadlock condition.

The single-step exception (XCPT_SINGLE_STEP) is not lost if the single-step operation causes a notification to be sent to DosDebug. In this case, the single-step exception is queued.

The single-step operation is not lost if other notifications were queued before the DBG_C_SStep command was issued. The Debug DBG_C_Continue command will clear the notifications one at a time until DosDebug has been completely notified. On the last DBG_C_Continue command, the single-step operation will take place as originally requested.

When a single-step operation is interrupted by an exception, the EIP (instruction pointer) should be moved to the next RING3 instruction. This may be in ring 3 system code. The single-step notification will be issued at this time.

The DBG_C_SStep command correctly single-steps most instructions. Single-stepping some REP instructions may not work correctly due to errors in the 80386 processor.

DBG_C_Stop

Debug Command 10 - Stop Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_Stop

Returns

The function performed by this command depends on the current state of the debuggee process, as follows:

- If the debuggee is already stopped:

 If there is a pending notification from the current thread, it is returned. See DosDebug Notifications for information about pending notifications.

 If there is no pending notification from the current thread, DBG_N_Success is returned.

- If the debuggee is executing user code:

 The debuggee is marked to stop before the next time it is ready to execute user-space (ring 2 or 3) code. This is known as an asynchronous stop.

 Kernel operations will not be interrupted for this DBG_C_Stop That is, threads blocked in the kernel (via a semaphore or internal operation) will not be interrupted. However, an infinite loop in user space will be stopped.

 Note: The asynchronous variation of the stop command implies a debugger with a minimum of two threads; one waits for a DBG_C_Go or DBG_C_SStep command to finish, and another executes the DBG_C_Stop command.

DBG_C_Freeze

Debug Command 11 - Freeze Thread Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of thread to freeze
Cmd = DBG_C_Freeze

If Tid is zero, all debuggee threads will be frozen.

Returns

The desired threads are prevented from executing user code on a DBG_C_Go or DBG_C_SStep command.

By using the DBG_C_Freeze and DBG_C_Resume commands, a given set of threads can be executed at once, while keeping the other threads suspended.

No error is returned if the thread was previously frozen; it just remains frozen. DBG_C_Freeze and DBG_C_Resume commands cannot be nested.

If the Tid is specified as zero, it will be set to the thread ID of the debuggee thread most recently scheduled to execute.

DBG_C_Resume

Debug Command 12 - Resume Thread Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of thread to thaw
Cmd = DBG_C_Resume

If Tid is zero, all debuggee threads will be thawed.

Returns

The DBG_C_Resume command complements the DBG_C_Freeze command. A thread that has been resumed will function as if it were never frozen.

No error is returned if the thread was not previously frozen.

If the Tid is specified as zero, it will be set to the thread ID of the debuggee thread most recently scheduled to execute.

DBG_C_NumToAddr

Debug Command 13 - Convert Object Number to Address Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_NumToAddr
Value = Logical object number in module
MTE = Module handle of module of interest

Returns

Addr = Starting address of object
Value = Logical object number
MTE = Module handle of module of interest

The logical object number in Value is converted into an address that points to the starting address of the desired logical object of the specified module in the debuggee's memory space. This address is then stored in the Addr field, in the form of a linear address.

The Value and MTE fields are left unchanged.

The logical object numbers for a module are generated at link time. By using this function, a debugger can discern the relationship between addresses and logical object numbers. Once the logical object number is known, symbols can be generated for an address via a map or symbol file, for symbolic debugging.

DBG_C_ReadCoRegs

Debug Command 14 - Read Coprocessor Registers Command

Parameters

Pid = Process ID of debuggee processor
Tid = Thread ID of Coprocessor register set to read
Cmd = DBG_C_ReadCoRegs
Value = Coprocessor Type Identifier
Buffer = Pointer to Coprocessor Register Context Buffer
Len = Size of Coprocessor Register Context Buffer
Index = Reserved, must be zero

If Tid is zero and the debuggee is stopped, the register set comes from the active debuggee thread. If Tid is zero and the debuggee is executing, ERROR_INVALID_THREADID is returned.

The coprocessor type identifier is a number that identifies the format of the coprocessor register context buffer. The buffer length must correspond exactly to the requested buffer format. The supported coprocessor types, formats and lengths include the following:

For the Intel 80387 NPX processor:

Value = DBG_CO_387 = 1
Len = 108

The coprocessor register context buffer format is the same as that defined by the fsave/frestore instructions as executed by the appropriate processor.

Returns

The debugger's coprocessor register context buffer is filled in with a copy of the registers read from the appropriate coprocessor, for the

thread specified in the Tid field.

If an error occurs while attempting to access the coprocessor context during this command, the DBG_N_CoError notification is returned.

Restrictions

An error is returned if any one of the following occurs:

- The debuggee process is emulating the coprocessor.
- The specified debuggee thread has not yet attempted to use the coprocessor.
- The wrong coprocessor type is used.
- Index is not zero.

DBG_C_WriteCoRegs

Debug Command 15 - Write Coprocessor Registers Command

Parameters

Pid = Process ID of debuggee processor
Tid = Nonzero Thread ID of Coprocessor register set to read
Cmd = DBG_C_WriteCoRegs
Value = Coprocessor Type Identifier
Buffer = Pointer to Coprocessor Register Context Buffer
Len = Size of Coprocessor Register Context Buffer
Index = Reserved, must be zero

The coprocessor type identifier is a number that identifies the format of the coprocessor register context buffer. The buffer length must correspond exactly to the requested buffer format.

See DBG_C_ReadCoRegs for the supported coprocessor types, formats, and lengths.

The coprocessor register context buffer format is the same as that defined by the fsave/frestore instructions as executed by the appropriate processor.

Returns

The debuggee thread's coprocessor registers are filled with the values passed via the coprocessor register context buffer, for the thread specified in the Tid field.

If an error occurs while attempting to access the coprocessor context during this command, the DBG_N_CoError notification is returned.

Restrictions

An error is returned if any one of the following occurs:

- The debuggee process is emulating the coprocessor.
- The specified debuggee thread has not yet attempted to use the coprocessor.
- The wrong coprocessor type is used.
- Index is not zero.

The coprocessor may adjust some control register bits, but DosDebug will not return an error if a modification is attempted, nor will it mask the values. Because of internal coprocessor management, this adjustment may be delayed until the thread actually uses the coprocessor again.

DBG_C_ThrdStat

Debug Command 17 - Get Thread Status Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of thread of interest
Cmd = DBG_C_ThrdStat
Buffer = Pointer to Thread Status buffer
Len = Length of Thread Status buffer, in bytes. This value is 4.

If Tid is zero, the status of the debuggee thread most recently scheduled to run will be returned.

Returns

Value	= Thread ID of "next" active thread to examine
Tid	= Thread ID of thread whose status is returned
Thread Status buffer	= Buffer for the thread status, filled in.

Thread Status buffer format is as follows:

```
typedef struct _TStat {
  BYTE      DbgState;
  BYTE      TState;
  USHORT    TPriority;
} TStat
```

DbgState in the Thread Status buffer contains information about the current state of debugging, and will have one of the following values upon return:

0	DBG_D_Thawed
1	DBG_D_Frozen

TState in the Thread Status buffer contains information about the scheduling state of the thread, and will have one of the following values upon return:

0	DBG_T_Runnable
1	DBG_T_Suspended
2	DBG_T_Blocked
3	DBG_T_CritSec

TPriority in the Thread Status buffer contains the thread's base scheduling priority. This priority will be expressed as scheduling class and delta values upon return.

The Value field will be filled in with the Thread ID of the "next" thread to look at when traversing threads.

By repeatedly calling the DBG_C_ThrdStat command, replacing the Tid with the last returned Value until a thread ID is repeated, all threads in the process can be traversed. When used in this way, the Tids returned by the DBG_C_ThrdStat command form a loop of the debuggee's thread IDs.

DBG_C_MapROAlias

Debug Command 18 and 19 - Map Read-Only or Read-Write Memory Alias Command

Parameters

Pid	= Process ID of debuggee
Cmd	= DBG_C_MapROAlias (Read Only) (*Not always supported*)
Cmd	= DBG_C_MapRWAlias (Read Write)
Buffer	= Reserved, must be zero.
Addr	= Start of debuggee region to alias (Page-Aligned)
Len	= Requested length of alias region (Page-Multiple)

Returns

Buffer	= Address of the start of the debugger alias region

An alias to the debuggee's memory region of the requested length is mapped into the debugger's memory space. This region is reserved for use as an alias region until it is unmapped.

The access rights for the alias area are determined by the command number. The DBG_C_MapROAlias command maps a read-only alias region, while the DBG_C_MapRWAlias command maps a read-write alias region.

For read-write aliases, if the region is shared and read-only in the debuggee's context, a private copy of the aliased pages will be created in the debuggee's context, and dynamic RAS tracepoints will be removed from that region. This prevents debugging from affecting other areas of the system, while allowing access to shared memory areas, and proper disassembly of regions where dynamic RAS tracepoints are in use.

Because the read-write aliases may convert objects to private, using up system resources, it is recommended that read-only aliases be used

when simply perusing memory. See the following **Restrictions** regarding read-only aliases on the 80386 processor.

Because the entire aliased region may map both valid and invalid regions of memory, debuggers should issue DosQueryMem just before accessing the alias region to determine if the region is valid. Debuggers should not access this region while the debuggee is executing, as portions of this region may become invalid without notifying the debugger. It is possible that no valid pages will exist in the alias region.

When the debuggee frees an aliased object, or shrinks the underlying object such that the alias would span a region outside the resultant object, an alias-free notification is returned to the debugger. This notification will be returned before the alias is invalidated. See DosDebug Notifications for details.

These commands may be performed while the debuggee is executing code via a DBG_C_Go command.

Restrictions

Because debuggers can execute code at ring 2, and the read-only bit in the page tables entries is effective only at ring 3, the read-only aliases cannot be supported. When the read-only bit becomes effective at all rings, as is expected on later processors, the read-only aliases will again be supported.

Most memory management calls may not be used on these aliases. DosQueryMem is permitted, but for interrogation only.

The passed starting addresses must be aligned on a page boundary, and the length of the aliased region must be a multiple of the page size. This restriction is due to the underlying hardware.

Aliased regions must be completely contained within a single debuggee memory object.

No interface is available for moving an alias to point to another section of debuggee memory. To move an alias, the debugger must free an existing alias, and then map a new alias to the desired region.

Aliases will only be permitted to the user space memory region of the debuggee. No aliases will be provided to system space.

The alias region will only be provided at the linear level. No debugger Local Descriptor Table (LTD) selector will be available to access the alias region.

DBG_C_MapRWAlias

Debug Command 18 and 19 - Map Read-Only or Read-Write Memory Alias Command

Parameters

Pid	= Process ID of debuggee
Cmd	= DBG_C_MapROAlias (Read Only) (*Not always supported*)
Cmd	= DBG_C_MapRWAlias (Read Write)
Buffer	= Reserved, must be zero.
Addr	= Start of debuggee region to alias (Page-Aligned)
Len	= Requested length of alias region (Page-Multiple)

Returns

Buffer	= Address of the start of the debugger alias region

An alias to the debuggee's memory region of the requested length is mapped into the debugger's memory space. This region is reserved for use as an alias region until it is unmapped.

The access rights for the alias area are determined by the command number. The DBG_C_MapROAlias command maps a read-only alias region, while the DBG_C_MapRWAlias command maps a read-write alias region.

For read-write aliases, if the region is shared and read-only in the debuggee's context, a private copy of the aliased pages will be created in the debuggee's context, and dynamic RAS tracepoints will be removed from that region. This prevents debugging from affecting other areas of the system, while allowing access to shared memory areas, and proper disassembly of regions where dynamic RAS tracepoints are in use.

Because the read-write aliases may convert objects to private, using up system resources, it is recommended that read-only aliases be used when simply perusing memory. See the following **Restrictions** regarding read-only aliases on the 80386 processor.

Because the entire aliased region may map both valid and invalid regions of memory, debuggers should issue DosQueryMem just before accessing the alias region to determine if the region is valid. Debuggers should not access this region while the debuggee is executing, as portions of this region may become invalid without notifying the debugger. It is possible that no valid pages will exist in the alias region.

When the debuggee frees an aliased object, or shrinks the underlying object such that the alias would span a region outside the resultant object, an alias-free notification is returned to the debugger. This notification will be returned before the alias is invalidated. See DosDebug Notifications for details.

These commands may be performed while the debuggee is executing code via a DBG_C_Go command.

Restrictions

Because debuggers can execute code at ring 2, and the read-only bit in the page tables entries is effective only at ring 3, the read-only aliases cannot be supported. When the read-only bit becomes effective at all rings, as is expected on later processors, the read-only aliases will again be supported.

Most memory management calls may not be used on these aliases. DosQueryMem is permitted, but for interrogation only.

The passed starting addresses must be aligned on a page boundary, and the length of the aliased region must be a multiple of the page size. This restriction is due to the underlying hardware.

Aliased regions must be completely contained within a single debuggee memory object.

No interface is available for moving an alias to point to another section of debuggee memory. To move an alias, the debugger must free an existing alias, and then map a new alias to the desired region.

Aliases will only be permitted to the user space memory region of the debuggee. No aliases will be provided to system space.

The alias region will only be provided at the linear level. No debugger Local Descriptor Table (LDT) selector will be available to access the alias region.

DBG_C_UnMapAlias

Debug Command 20 - UnMap Memory Alias Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_UnMapAlias
Buffer = Address of the debugger alias region to unmap

Returns

The DBG_C_UnMapAlias command is used when the debugger has finished using an alias region. Both read-only and read-write aliases may be freed in this way.

Regions returned from other memory management calls may not be used.

The debugger may issue this command while the debuggee is executing code via a DBG_C_Go command.

When the debuggee process terminates, all aliases to its memory space will be invalidated. When a debugger program terminates, all aliases from its memory space will also be invalidated.

DBG_C_Connect

Debug Command 21 - Connect To Debuggee Command

Parameters

Addr = Possible values are shown in the list below:

0x00000000	The default action is to sever the connection between the debugger and the program being debugged if a system resource is being held.
0x00000001	The Sever action is not wanted between the debugger and the program being debugged.

Pid = Process ID of debuggee

Tid = Reserved, must be zero

Cmd = DBG_C_Connect

Value = Debugging Level Number

The only permitted debugging level number is shown in the following list:

1 = DBG_L_386

This must be the first DosDebug command. No other DosDebug command will be accepted until the debugging connection has been established.

Returns

This command establishes a debugging connection. It must be the initial command, since it verifies the buffer format for the rest of the connection.

Because DosDebug usually cannot be ported to new machines without changing the format of the buffer, this command is needed to establish that the debugger is capable of handling the desired buffer format.

If the requested debugging level is not supported, an error is returned, and the connection is not made. This gives the debugger a chance to try again, or to automatically start a different debugger process that uses a different buffer format.

For this command, only the machine-independent portion of the buffer is examined. This portion includes the Pid, Tid, Cmd, and Value fields. This makes it possible to port the DosDebug buffer from one machine to another, without returning an error to the debugger on the initial DosDebug command.

The only DosDebug notifications that are returned by this command are DBG_N_Success and DBG_N_Error.

Restrictions

If the connection to the debuggee is not established within a reasonable amount of time, it is assumed that the connection will never be established, and the debuggee process is terminated automatically.

The current format level may or may not be supported in future versions. This is due to the machine dependence of the DosDebug function.

Remarks

If *Addr* is set to 0, the connection between the debugger and the program being debugged is not severed. If any debuggee threads, other than the thread that encountered the debug event, are holding system semaphores, they will be allowed to run until they release the semaphores. They will then be stopped and the notification will be delivered.

If the thread encountering the debug event is holding a system semaphore the debugger/debuggee connection is severed by terminating the debuggee, and returning a DBG_N_Error notification to the debugger with the value field set to 0, and the register set filled in. No further DosDebug commands will be accepted by the debuggee, nor will it generate any other notifications.

If a DBG_C_Stop is issued, and a thread owning a system semaphore is about to generate a DBG_N_AsyncStop notification, it will be allowed to continue execution until it releases the semaphore. It will then be stopped, and the notification delivered. This is the only exception to the severing of the debugger/debuggee rule.

If *Addr* is set to 0, the connection between the debugger the program being debugged is severed if a system resource is being held,. in which case DosDebug returns:

Tid = Thread owning semaphore
Cmd = DBG_N_Error
Value = ERROR_EXCL_SEM_ALREADY_OWNED

If the debugger needs to present some information to the user or use the thread holding the system resource, the debugger must terminate the program being debugged. Any other action might result in a system halt.

DBG_C_ReadMemBuf

Debug Command 22 - Read Memory Buffer Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_ReadMemBuf
Addr = Debuggee address to read from
Buffer = Debugger address to copy to
Len = Number of bytes to read

Returns

The number of bytes specified by Len is copied from the debuggee's user memory space starting at *Addr* into the debugger's Buffer.

This command is not serialized with respect to the DBG_C_Go command.

Restrictions

You are unable to read from any memory outside user space.

Both specified memory regions must be currently valid.

DBG_C_WriteMemBuf

Debug Command 23 - Write Memory Buffer Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_WriteMemBuf
Addr = Debuggee address to write to
Buffer = Debugger address to copy from
Len = Number of bytes to write

Returns

The number of bytes specified by Len is copied from the debugger's Buffer into the debuggee's memory space starting at Addr.

This command is not serialized with respect to the DBG_C_Go command.

In the case of a write to shared read-only memory, the memory is first converted to private, and any set dynamic RAS logging points are removed from that memory, before the write is performed.

Dynamic RAS logging will continue to function in that area, in the context of other processes. The area will continue to be shared by other processes, if any.

In this way, breakpoints may be set in the debuggee without affecting the other modules.

Restrictions

You are unable to write to any memory outside user space.

Both specified memory regions must be currently valid.

DBG_C_SetWatch

Debug Command 24 - Set Watchpoint Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_SetWatch
Addr = Starting Address of Watchpoint
Len = Length of Watchpoint, in bytes
Index = Reserved, must be zero
Value = Watchpoint Type and Scope

The Watchpoint Type and Scope is a combination of a Scope number and a Type number. Both the Scope and Type must be specified. For example, to set a local watchpoint for either read or write access, Value should be set to (DBG_W_Local + DBG_W_ReadWrite).

The Watchpoint Scopes are:

DBG_W_Global (00000001h)
DBG_W_Local (00000002h)

The Watchpoint Types are:

DBG_W_Execute (00010000h)
DBG_W_Write (00020000h)

DBG_W_ReadWrite (00030000h)

Returns

Index = Watchpoint ID Number

This command sets a code or data watchpoint of the desired scope and type to cover the specified range of addresses.

The Watchpoint Scope controls the context in which the watchpoint is actually effective. DBG_W_Local watchpoints are effective only in the context of the debuggee process, while DBG_W_Global watchpoints are effective in the context of any process.

Both DBG_W_Local and DBG_W_Global watchpoints remain effective at interrupt time, and while executing kernel code. However, the DBG_W_Local watchpoints may miss interrupt time accesses, depending on the process context in which the interrupt occurred.

Watchpoints are disabled as soon as they are hit, so that they can only be hit once.

The resources used by a watchpoint will not be freed until the debugger is finally notified of the hit, or the debugger terminates. The debugger should use the DBG_C_Stop command to free resources held by any pending watchpoint hits prior to setting a watchpoint, so that these held resources will not prevent setting a new watchpoint.

DBG_W_Global watchpoints should be used sparingly, as they restrict the watchpoint resources available to all processes at once. Watchpoint resources are very limited.

Restrictions

The watchpoints are restricted by the hardware. In the case of the 80386 processor, where debug registers are used, the available watchpoint lengths are 1, 2, and 4 bytes. The 2-byte data watchpoints must be aligned on a word boundary, and the 4-byte data watchpoints must be aligned on a doubleword boundary. DBG_W_Execute watchpoints must be exactly 1 byte in length, and they must begin on an instruction boundary to be effective.

Global watchpoints are effective in v86 mode, but cannot detect DMA (direct memory access) device accesses.

Global watchpoints may be set only in the shared memory region of the linear address space. Global watchpoints will remain effective even if the underlying memory has been converted to private memory via a DosDebug memory write operation.

DBG_C_ClearWatch

Debug Command 25 - Clear Watchpoint Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_ClearWatch
Index = Watchpoint ID Number

Returns

This command clears a currently set or hit watchpoint.

If the watchpoint is not currently set, an error is returned.

If a debugger wishes to move a watchpoint from one location to another, it should clear the old watchpoint before setting the new one, to free any resources used by currently set watchpoints.

The watchpoint will be cleared even if it is currently hit, and a notification is pending. To prevent missing the watchpoint hit in this way, you should issue the DBG_C_Stop command just before clearing the watchpoint, to pick up any pending watchpoint hit notifications.

DBG_C_RangeStep

Debug Command 26 - Range Step Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of thread to range-step
Cmd = DBG_C_RangeStep

Value = Linear address denoting start of range (exclusive)
Addr = Linear address denoting end of range (exclusive)

Returns

The RangeStep notification is usually returned, but any Debug notification may be returned. See DosDebug Notifications for more information.

This command allows a debugger to specify a range of addresses (bounded by the linear addresses in the Value and Addr fields) through which a debuggee thread should single-step until one of the following conditions occurs:

- The debuggee thread's linear EIP (instruction pointer) is outside the range.
- The linear EIPs of consecutive debuggee threads are the same.
- Some other notification occurs.

When the DBG_C_RangeStep command returns, the register set is automatically updated to reflect the thread that detected the event.

Callgates that result in a privilege level transition to ring 0 will appear to range-step as a single instruction, with the range-step continuing after the function completes. This hides ring 0 execution from debuggers.

Attempting to range-step a thread that is frozen results in an error.

Restrictions

To accomplish callgate single-stepping, the single-step must be simulated because the flags (specifically, the TF bit) are not stored in the ring 0 callgate stack frame. Because of this, a range-step that results in leaving a ring 0 callgate will sometimes not execute any user-space code. The following range-step should function normally.

Range-stepping some REP instructions may not work correctly due to errors in the 80386 processor.

DBG_C_Continue

Debug Command 27 - Continue After an Exception Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID
Cmd = DBG_C_Continue
Value = XCPT_CONTINUE_EXECUTION (0xFFFFFFFF), or XCPT_CONTINUE_SEARCH (0x00000000), or XCPT_CONTINUE_STOP (0x00716668)

Returns

You must issue the DBG_C_Continue command to continue after DosDebug has been given preemptive notifications or an exception notification. For such notifications, the DBG_C_Continue command is the only Debug command that will start the child process again. You can issue other Debug commands, but you must eventually issue the DBG_C_Continue command.

If you issue the DBG_C_Continue command and there is no pre-existing notification or exception, the DBG_C_Continue command acts like a Debug DBG_C_Go.

In single-step mode, XCPT_CONTINUE_STOP has the same effect as XCPT_CONTINUE_EXECUTION. That is, execution is always stopped after a single-step operation when DBG_N_Success is returned.

Handling Preemptive Notifications

The DBG_C_Continue command is used to either continue or stop the child process after a preemptive notification has been received from DosDebug.

The XCPT_CONTINUE_STOP parameter can be used to stop the child process after a preemptive notification has been received. Any pending notifications will be held until execution of the child process is resumed using subsequent DosDebug commands. While the child process is stopped, you can issue other DosDebug commands, such as DBG_C_ReadMem.

The XCPT_CONTINUE_SEARCH parameter allows the child process to execute until the next notification is received.

The following is a list of preemptive notifications.

- DBG_N_ModuleLoad
- DBG_N_ModuleFree
- DBG_N_NewProc
- DBG_N_ProcTerm
- DBG_N_ThreadCreate

- DBG_N_ThreadTerm
- DBG_N_AliasFree
- DBG_N_Exception

Handling the DBG_N_Exception Notification

Note: XCPT_BREAKPOINT and XCPT_SINGLE_STEP are pre-first chance exception notifications.

The XCPT_CONTINUE_STOP parameter serves two purposes. It stops the child process, and it tells DosDebug that the debugger handled the exception.

The XCPT_CONTINUE_EXECUTION parameter tells DosDebug to restore the execution context of the thread that received the exception, and then continue execution of the child process. This implies that the debugger has handled the exception.

The XCPT_CONTINUE_SEARCH parameter tells DosDebug to pass the exception to the exception handler because the debugger will not handle it. After receiving an exception notification other than XCPT_BREAKPOINT or XCPT_SINGLE_STEP, the DBG_C_Continue command with the XCPT_CONTINUE_SEARCH parameter resumes execution of the child process.

DBG_C_AddrToObject

Debug Command 28 - Get Memory Object Information Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_AddrToObject
Addr = Debuggee Linear Address

Returns

Buffer = Starting address of object
Len = Size of object in bytes
Value = Object flags
MTE = Module Table Entry handle of module if DBG_O_OBJMTE is set in the Value field object flags.

The object flags are defined as follows:

DBG_O_OBJMTE (10000000h) - Object is part of a module

Information about the memory object containing the linear address is returned. If the linear address is not part of an object, DBG_N_Error is returned with ERROR_INVALID_OBJECT in the Value field.

The Addr field will be left unchanged.

DBG_C_XchngOpcode

Debug Command 29 - Exchange Opcode and Go Command

Parameters

Pid = Process ID of debuggee
Tid = Thread ID of thread
Cmd = DBG_C_XchngOpcode
Value = Opcode 1 for Single Step
Addr = Opcode 2 for Go

Returns

The sequence of operations for this Debug command is:

1. Replace the code at the EIP (instruction pointer) with opcode 1.

2. Single-step the thread specified by the Tid field. Do not execute other threads. If the single-step operation goes into ring 0 code, consider the single-step operation complete at the first ring 0 instruction.

3. Replace the code at the original EIP with opcode 2.

4. Issue a Debug Go command on all non-frozen threads.

If an exception that DosDebug is to be notified about occurs during the single-step operation of this Debug command, opcode 2 is placed at the original EIP, and DosDebug is notified of the exception. When the debugger issues the Debug Continue command, the child process continues execution.

Note: If an exception that DosDebug is *not* to be notified about occurs, then the DBG_C_XchngOpcode command executes as if no exception took place.

If opcode 1 and opcode 2 are identical, this Debug command executes only operations 3 and 4 above. There is no need to single-step the thread specified by the Tid field. This would be a "replace opcode and go" sequence using only one DosDebug function instead of two.

DBG_C_LinToSel

Debug Command 30 - Translate Linear Address to Selector:Offset Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_LinToSel
Addr = Linear address to be translated

Returns

Value = Selector
Index = Offset

The Addr field will be left unchanged.

DBG_C_SelToLin

Debug Command 31 - Translate Selector:Offset to Linear Address Command

Parameters

Pid = Process ID of debuggee
Cmd = DBG_C_SelToLin
Value = Selector from address to be translated
Index = Offset from address to be translated

Returns

Addr = Linear address

The Value and Index fields will be left unchanged.

DosDebug Notifications

For a description of the data returned with each notification, select

Not. No.	Notification Name	Description
0	DBG_N_Success	Successful command completion
-1	DBG_N_Error	Error detected during command
-6	DBG_N_ProcTerm	Process termination - DosExitList done

-7	DBG_N_Exception	Exception detected
-8	DBG_N_ModuleLoad	Module loaded
-9	DBG_N_CoError	Coprocessor not in use error
-10	DBG_N_ThreadTerm	Thread termination - not in DosExitList
-11	DBG_N_AsyncStop	Async Stop detected
-12	DBG_N_NewProc	New Process started
-13	DBG_N_AliasFree	Alias needs to be freed
-14	DBG_N_Watchpoint	Watchpoint hit
-15	DBG_N_ThreadCreate	Thread creation
-16	DBG_N_ModuleFree	Module freed
-17	DBG_N_RangeStep	Range Step detected

Note: References to "IP" in the data return descriptions refer to the instruction pointer address. This is the 32-bit equivalent of the CS:EIP instruction pointer, regardless of the CS selector. This is also known as a linearized instruction pointer.

Some notifications (such as DBG_N_ModuleLoad and DBG_N_Watchpoint) may require multiple returns to the debugger. These additional pending notifications will be returned before the process being debugged can execute any more user code, and will be returned on the DBG_C_Go, DBG_C_SStep, or DBG_C_Stop commands.

Note that more notifications might be pending at any time, so a debugger should be ready to handle any notification at any time that a DBG_C_Go, DBG_C_SStep, or DBG_C_Stop command is issued.

If DosDebug returns ERROR_INTERRUPT after a command, the next notification might have been lost. If the process being debugged was executing code at that time (via a DBG_C_Go, DBG_C_SStep or DBG_C_RangeStep command), it will be stopped automatically. To prevent this, DosDebug should not be used by thread 1 while signals are being used, or the debugger should issue DosEnterMustComplete before issuing the command.

DBG_N_Success

Debug Notification 0 - Success Notification

This notification returns:

Cmd = DBG_N_Success

The DosDebug command was successful. Returned values depend on the command just executed.

DBG_N_Error

Debug Notification -1 - Error Notification

This notification returns:

Cmd = DBG_N_Error
Value = Any standard error code

An error was detected while attempting a DosDebug command.

Error codes are returned from DosDebug in different ways:

1. Errors can be returned via the standard method (EAX = ERROR_CODE).

 This is only done when the DosDebug command failed to execute at all. An example of this would be ERROR_INTERRUPT when the debugger receives a signal. If DosDebug returns a nonzero value in the EAX register, the returned Debug buffer is invalid.

2. Errors are also returned via the DBG_N_Error notification.

 This is used when the DosDebug command was attempted, but failed due to some internal DosDebug failure. Examples of error codes returned via this interface are ERROR_INVALID_PROCID and ERROR_INVALID_DATA. Generally, these errors are detected while in the context of the debuggee process, and may include ERROR_INTERRUPT.

DBG_N_ProcTerm

Debug Notification -6 - Process Termination Notification

This notification returns:

Cmd = DBG_N_ProcTerm
Value = Process Exit Code
Index = Process Exit Type
Addr = 0

The debuggee process is about to terminate.

The debugger is still allowed to examine the debuggee's final register and memory contents at this time. Note that when the debugger has completed this examination, it should finish terminating the debuggee process with a final DBG_C_Go, DBG_C_SStep, or DBG_C_Term command.

Value and Index contain the same information as that returned via a subsequent DosWaitChild call. The act of collecting this information does not terminate the process.

DBG_N_Exception

Debug Notification -7 - General Exception Notification

This notification returns:

- For the pre-first chance for a breakpoint exception:

 Cmd = DBG_N_Exception
 Value = 0 (DBG_X_PRE_FIRST_CHANCE)
 Addr = Linear address of breakpoint
 Buffer = Exception number (XCPT_BREAKPOINT) (0xC000009F)

- For the pre-first chance for a single-step exception:

 Cmd = DBG_N_Exception
 Value = 0 (DBG_X_PRE_FIRST_CHANCE)
 Addr = Linear address of instruction after Single Step
 Buffer = Exception number (XCPT_SINGLE_STEP) (0xC00000A0)

- For the first chance for all exceptions:

 Cmd = DBG_N_Exception
 Value = 1 (DBG_X_FIRST_CHANCE)
 Addr = Linear address of exception
 Buffer = Pointer to Exception Report Record in debuggee's context
 Len = Pointer to Exception Context Record in debuggee's context

- For the last chance for all exceptions:

 Cmd = DBG_N_Exception
 Value = 2 (DBG_X_LAST_CHANCE)
 Addr = Linear address of exception
 Buffer = Pointer to Exception Report Record in debuggee's context
 Len = Pointer to Exception Context Record in debuggee's context

- For an invalid stack notification:

Cmd = DBG_N_Exception
Value = 3 (DBG_X_STACK_INVALID)
Addr = Linear address of exception
Buffer = Exception number

The scenarios under which a debug exception is reported are pre-first, first, and last chance, and invalid stack notification. The Value field of the user debug buffer indicates the scenario.

DosDebug has detected an exception (a trap or a fault) at the specified address. The exception number in the exception structure identifies the exception that was detected.

Exception notifications are always returned from the context of the thread that detected the exception. That is, the exception structure reflects the state of the thread that caused the exception, at the time the exception was detected.

The debugger is given a maximum of two chances to handle exceptions other than single-step or breakpoint exceptions, which have a maximum of three chances. The order of operations for handling an exception is as follows:

1. Debugger has the pre-first chance to handle the exception (for breakpoint and single-step exceptions).
2. Debugger has the first chance to handle the exception, or to invoke an exception handler if it is present.
3. Debugger has the last chance to handle the exception, or to invoke an exception handler if it is present.

An exception notification is returned for all exceptions, including those raised by the user via DosRaiseException.

An exception can have an informational, warning, or fatal severity. The severity is coded in the high-order three bits of the exception number for user-raised and system exceptions.

The debugger may dismiss the exception by returning XCPT_CONTINUE_EXECUTION, so that the user's context is restored, and execution continues at the point where the exception occurred. Otherwise, the debugger may return XCPT_CONTINUE_SEARCH. This causes the exception to be passed to the user's exception handlers (after the debugger's first chance), or causes the default action for the exception to occur (after the debugger's last chance).

For performance reasons, the single-step and breakpoint exceptions cause a "pre-first" notification. This is faster than the ordinary first exception notification. At the time of the notification, the debugger may decide if the single-step or breakpoint exception was an anticipated event. If it was anticipated, the debugger may return XCPT_CONTINUE_EXECUTION, as for an ordinary first notification. If it was not anticipated, the debugger may return XCPT_CONTINUE_SEARCH in order to raise an ordinary first notification for the single-step or breakpoint exception. With the second notification, this allows a maximum of three notifications for the single-step and breakpoint exceptions.

For breakpoint exceptions, the instruction pointer (EIP) of the debuggee is decremented to point to the breakpoint instruction.

Note: Do not confuse the family of floating point exceptions with the DBG_N_CoError error notification.

Restrictions

The error code may not be reliable in some situations for the page fault exception, due to hardware errors.

DBG_N_ModuleLoad

Debug Notification -8 - Module Load Notification

This notification returns:

Cmd = DBG_N_ModuleLoad
Value = MTE (Module Table Entry) handle of newly attached module
Addr = 0

A module has just been loaded. This occurs either at program startup, or during a call to DosLoadModule.

The newly attached module's handle is returned via Value. You can use this handle with DosQueryModuleName, or with the Debug DBG_C_NumToAddr command, for symbolic debugging. A debugger should save these handles for future reference.

There may be many module attachments done at one time, but DosDebug is only able to communicate a single load during any one notification. In this case, the additional library load notifications become pending. The debugger should issue repeated DBG_C_Stop commands to be notified of these additional library loads, until Success is returned from the DBG_C_Stop command. If the DBG_C_Go, DBG_C_SStep, or DBG_C_RangeStep commands are issued instead of the DBG_C_Stop command, the pending notifications will be returned immediately, until there are no further notifications.

DBG_N_CoError

Debug Notification -9 - Coprocessor Error Notification

This notification returns:

Cmd = DBG_N_CoError
Value = Any standard Error Code

An error was detected while attempting a DosDebug command that attempted to access a Coprocessor.

DBG_N_CoError is similar to the DBG_N_Error notification, but is returned only after attempting an access to the coprocessor registers.

DBG_N_CoError is returned if any one of the following occurs:

- The debuggee process is emulating the coprocessor.
- The specified debuggee thread has not yet attempted to use the coprocessor.
- The wrong coprocessor type is used.
- The requested coprocessor type is not supported or available.

This notification should not be confused with any of the floating point exception notifications.

DBG_N_ThreadTerm

Debug Notification -10 - Thread Termination Notification

This notification returns:

Cmd = DBG_N_ThreadTerm
Value = Thread's proposed return code (from DosExit)
Addr = 0

A debuggee thread is about to terminate.

DosExitList processing has not yet been started.

The debugger is still allowed to examine the debuggee thread' signal register contents at this time. When the debugger has completed this examination, it should finish terminating the debuggee thread with a final DBG_C_Go or DBG_C_SStep command.

Value contains the return code from the thread. This is only a proposed return code passed from DosExit. Only when the process actually terminates is the return code that is passed to DosWaitChild finally known.

DBG_N_AsyncStop

Debug Notification -11 - Asynchronous Stop Notification

This notification returns:

Cmd = DBG_N_AsyncStop
Value = 0
Addr = 0

An Asynchronous Stop request has been detected.

The asynchronous stop command is used to get the attention of some debuggee thread, so that the debugger can again control the process. Because any notification results in the debuggee's coming under control of the debugger again, the asynchronous stop notification becomes redundant in that case, and is not returned.

The asynchronous stop request will be overridden and ignored if another notification can be performed instead, at the same time. An asynchronous stop notification never becomes a "pending" notification, in the sense that a library load notification becomes pending.

DBG_N_NewProc

Debug Notification -12 - New Process Notification

This notification returns:

Cmd = DBG_N_NewProc
Value = Process ID of the new process
Addr = 0

The debuggee process has just started a child process, and that child process needs to be debugged.

Note: This notification occurs only if descendant debugging was specified in the DosExecPgm call that started the process tree in which the debuggee is executing.

DBG_N_AliasFree

Debug Notification -13 - Alias Free Notification

This notification returns:

Cmd = DBG_N_AliasFree
Buffer = 32-bit offset to debugger alias region
Addr = 0

An object that contains an aliased region is about to be freed by the debuggee. This can also occur if the underlying memory object is about to be shrunk such that the alias would span memory outside the resultant object.

The alias region must be unmapped before the debugger may execute the debuggee again.

The DBG_C_UnMapAlias command is the proper response to an alias free notification.

If a debugger creates an alias to memory in another debugger, and that memory is itself an alias to the second debugger's debuggee, then the first debugger will not receive an alias free notification when the memory of the second debugger's debuggee is freed. The alias will be freed automatically. The second debugger will receive an alias free notification.

DBG_N_Watchpoint

Debug Notification -14 - Watchpoint Hit Notification

This notification returns:

Cmd = DBG_N_Watchpoint
Addr = Linearized instruction pointer of watchpoint hit
Value = Process ID of process that hit the watchpoint
Len = Thread ID of thread that hit the watchpoint
MTE = Module Table Entry handle of process that hit the watchpoint
Index = Watchpoint ID number

A watchpoint has been hit. The Watchpoint ID number identifies the watchpoint that was hit.

Multiple watchpoint hits become pending notifications that are returned on subsequent DBG_C_Stop, DBG_C_Go, or DBG_C_SStep commands. A watchpoint may be hit at any time, and more than one watchpoint may be hit at the same time.

If a watchpoint notification is pending, the resources used by the watchpoint will not be freed until the watchpoint notification is complete, or the watchpoint is cleared.

A watchpoint notification is not always returned by the same thread that caused the hit. A watchpoint may be hit by one thread, but another thread may return the notification. The thread ID of the thread that hit the watchpoint is not necessarily the value passed in the Tid field.

Data Watchpoint hits are treated as faults, rather than as traps, by the 80386 processor. Therefore, the linearized instruction pointer may not point to the exact instruction that caused the fault.

If a watchpoint is hit at interrupt time, the Value, Addr, MTE, and Len fields are all returned as zero.

DBG_N_ThreadCreate

Debug Notification -15 - Thread Creation Notification

This notification returns:

Cmd = DBG_N_ThreadCreate
Tid = Thread ID of new thread
Addr = 0

A debuggee thread has just been created.

The new thread has not executed any user code yet.

DBG_N_ModuleFree

Debug Notification -16 - Module Free Notification

This notification returns:

Cmd = DBG_N_ModuleFree
Value = MTE (Module Table Entry) handle of freed module
Addr = 0

A module has just been freed. This occurs either at program termination, or during execution of the DosFreeModule.

The newly attached module's handle is returned via Value. You can use this handle with DosQueryModuleName, or with the DosDebug DBG_C_NumToAddr command, for symbolic debugging. A debugger should save these handles for future reference.

There may be many modules freed at one time, but DosDebug is only able to communicate the freeing of a single module during any one notification. In this case, the additional notifications of freed modules become pending. The debugger should issue repeated DBG_C_Stop commands to be notified of these additional module freeing operations, until DBG_N_Error is returned from the DBG_C_Stop command. If the DBG_C_Go or DBG_C_SStep commands are issued instead of the DBG_C_Stop command, the pending notifications will be returned immediately, until there are no further notifications.

DBG_N_RangeStep

Debug Notification -17 - Range Step Notification

This notification returns:

Cmd = DBG_N_RangeStep
Addr = Linearized instruction pointer at exception
Value = Linearized instruction pointer of last user instruction executed

The debuggee stopped range-stepping because its linearized instruction pointer was outside the original range, or because the current linearized instruction pointer is equal to the linearized instruction pointer of the previous user instruction.

The DBG_N_RangeStep notification is returned independently of the DBG_N_Watchpoint notification, even though the Watchpoint fault and the RangeStep may have occurred at the exact same time.

Kernel Debugger Communications Protocol

The Kernel Debugger is essentially a replacement OS/2 Kernel module that contains a built-in debugger component. The kernel debugger can be used to halt system execution, inspect and alter memory and registers, and display system control blocks. The kernel debugger is

described in detail in *The OS/2 Debugging Handbook - Volume II*, IBM publication number SG24-4641.

Generally the kernel debugger communicates over a serial communications link with a terminal emulator program running on another machine. This allows a user to debug a problem by issuing kernel debugger commands in the emulator program and seeing the results displayed on the debug console.

To automate the debugging process, or to provide a high-level language debugging environment, this interaction with the kernel debugger could instead be handled by a program running on the other machine. This **debug engine** would interact with the user, convert the user's debugging request to a series of kernel debugger commands, issue them, and then present the response from the kernel debugger back to the user in a user-friendly format.

Communications between the debug engine and the kernel debugger can proceed in one of two modes:

raw (dumb TTY) mode
: ASCII characters are sent to the kernel debugger one at a time. The kernel debugger echoes each character. A carriage return (^M or 0x0d) ends a line. The kernel debugger returns data to the debug engine one ASCII character at a time.

packet mode
: Packets are sent to the kernel debugger. A packet consists of a fixed sized header followed by zero or more bytes of data. The kernel debugger returns data to the debug engine in packets.

Raw Mode

Any kernel debugger command may be sent in raw mode. Debug engines that communicate in packet mode may wish to use raw mode to issue a **.B** command to set the communication rate for the serial connection.

To enter raw mode from packet mode, or to get the kernel debugger's attention while the system is running and enter raw mode, the debug engine should send a break character (^C or 0x03) and wait for the kernel debugger to issue a prompt.

Packet Mode

To enter packet mode from raw mode, or to get the kernel debugger's attention while the system is running and enter packet mode, the debug engine should send the **KDP break** character (0x1f). If the system was running, the kernel debugger will respond with an event packet containing a CVK_RET_ASYNC event. If the system was quiesced, the kernel debugger will not respond at all.

Packet Format

The format of a packet is as follows:

```
                                          ...
0x1d   10-byte packet header   packet body (optional)   0x1e
                                          ...
```

The packet header and the packet body, if present, are **bitstuffed**. The data is treated as a stream of bits and is broken into seven-bit chunks. Each chunk is put into the seven low-order bits of a byte and the high order bit of the byte is set. The bitstuffed data is padded at the end with zero bits to the next byte boundary. For example, the header 00009540 0000ac57 when bitstuffed becomes 808092d4 808081ac abc0.

Packet Header Format

The header, prior to bitstuffing, contains a 4-byte logical ID field, a 2-byte length field, and a 2-byte checksum. The checksum of n bytes of data is computed as follows:

```
unsigned char  data[ ];
unsigned short checksum = 0xa1e8;
for (i = 0; i < n; i++)
{
  checksum += data[i];
  checksum  = (checksum << 3) + (checksum >> 13);
}
```

Packet Data Format

All multibyte items are presumed to appear in little-endian order. Thus, the checksum computed for the header 00009540 0000 is 0x57ac; when stored in the header the low order byte (0xac) appears first.

The header length field contains the number of bytes of data in the packet body before the packet body is bitstuffed. If the header length field is zero, there is no packet body. If a packet body is present, it includes a 2-byte checksum that is **not** accounted for in the header length field. For example, if the header length field is 0x12, there are actually 20 bytes in the unbitstuffed packet body.

The logical ID field takes one of the following forms (sequence numbers are one byte long):

```
CVK_HDR_DATA - 0x8000                          |
               0x4000 if "flast" flag is set   |
              ((index number & 0x3f) << 8) |
               sequence number

CVK_HDR_ACK - (0x4000 | ((index number & 0x3f) << 8) | sequence number) << 16

CVK_HDR_NACK- (0xc000 | ((index number & 0x3f) << 8) | sequence number) << 16
```

Maximum Packet Size

The maximum packet size for a bitstuffed packet is defined by CVK_PACKET_MAXSIZE as 623. This is derived by:

```
Start byte      1
Header         10
Data          611
End Byte        1
```

General Considerations

The debug engine initiates all transactions with the kernel debugger, normally by sending a command while the kernel debugger is waiting to receive one. In this case, the kernel debugger expects to receive a CVK_HDR_DATA packet and will discard any CVK_HDR_ACK or CVK_HDR_NACK packets it receives. (The kernel debugger also discards a packet if its header cannot be unbitstuffed or has a bad checksum. The kernel issues a CVK_HDR_NACK packet containing the sequence number and index number from the original CVK_HDR_DATA packet if the packet's body cannot be unbitstuffed or has a bad checksum.)

Once the kernel receives a valid CVK_HDR_DATA packet, it extracts the sequence number and index number and uses them to construct a

CVK_HDR_ACK packet, which it returns to the debug engine. (The **flast** flag in the CVK_HDR_DATA packet is ignored.) The kernel debugger then performs the action requested by the command and returns the result.

In general, the result is returned in a single CVK_HDR_DATA packet whose sequence number matches the sequence number contained in the debug engine's original command packet, whose index number is zero, and whose **flast** flag is FALSE. After the result is transmitted, the kernel debugger waits for a response from the debug engine. The kernel is expecting either a CVK_HDR_ACK packet whose sequence number and index number match those sent in the result _or_ a CVK_HDR_DATA packet (containing the next command). If the kernel debugger receives any other response, it resends the CVK_HDR_DATA packet containing the result.

If the debug engine sends a KDP break character while the victim machine is running, either to initiate a transaction or to regain control after the victim machine has resumed execution, the kernel debugger responds with a CVK_HDR_DATA packet whose sequence number matches the sequence number from the CVK_HDR_DATA packet that caused the system to resume. There is no such packet when the kernel debugger responds to the first KDP break sent by the debug engine. The sequence number in that case contains garbage.

The kernel debugger does not generate replies for some commands, such as reboot, and the replies to commands that cause the victim machine to resume execution, such as resume or step, are not sent until an event such as a breakpoint, module load, or break signal from the debug engine, has caused the victim machine to quiesce.

The kernel debugger responds somewhat differently to the CVK_CMD_RAW command, which is used to issue arbitrary kernel debugger commands while in packet mode. Each line in the response is returned in a separate CVK_HDR_DATA packet whose sequence number matches the sequence number in the CVK_CMD_RAW command's header. The index number in the first reply packet is 0; the index number increases by 1 in each successive reply packet (and wraps from 63 to 0). The debug engine should return a CVK_HDR_ACK packet with the appropriate sqeuence number and index number after each reply packet is received.

The kernel debugger does not manipulate or increment sequence numbers and uses them only to generate ACKs and NACKs and to match ACKs with replies. A debug engine could use the same sequence number for every request, but this is not recommended.

Kernel Debugger Packet Responses

Event	Code	Description
CVK_RET_SUC	0x0000	Success
CVK_BAD_COMMAND	0x0002	Unrecognized command
CVK_RET_PAGEIN	0xffef	Discarded page reloaded
CVK_RET_TEND	0xfff0	Task died
CVK_RET_TNEW	0xfff1	Task created
CVK_RET_ASYNC	0xfff5	Asynchronous halt (break)
CVK_RET_LIB	0xfff8	Module loaded
CVK_RET_GPF	0xfff9	General protection fault
CVK_RET_KIL	0xfffa	Module unloaded
CVK_RET_NMI	0xfffb	Non-maskable interrupt
CVK_RET_BPT	0xfffc	Software breakpoint (INT3)
CVK_RET_TBT	0xfffd	Single step
CVK_RET_ERR	0xffff	Failure

Events Reported by the Kernel Debugger

The following is a summary of the data reported when the kernel debugger sends an event in packet mode:

CVK_RET_GPF Events

Event	Code	Description
KDP_T_DIVIDE	0	Divide by zero exception
KDP_T_INTO	4	Overflow Interrupt (INTO instruction)
KDP_T_BOUND	5	Bounds check (BOUND instruction)
KDP_T_INVALID_OPCODE	6	Invalid operation
KDP_T_EXTENSION	7	Coprocessor not available
KDP_T_DOUBLE_EXCEPTION	8	Double exception
KDP_T_EXTENSION_SEG_OVERRUN	9	Coprocessor segment overrun
KDP_T_INVALID_TSS	10	Invalid TSS
KDP_T_SEG_NOT_PRESENT	11	Segment not present
KDP_T_STACK_SEG	12	Stack exception
KDP_T_GP_FAULT	13	General protection fault
KDP_T_PAGE_FAULT	14	Page fault

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_GPF
Value	Event code from the table above.
OffV	Linear address in CS:(E)IP at time of event.
SegV	Slot number of thread.
MTE	MTE entry of executable running in process.
PID	PID of process that generated the event.
TID	TID of thread that generated the event.
DBit	Flags from CS selector.
Reg	Registers at time of event.
MemCache	Not used.

CVK_RET_TBT Event

A CVK_RET_TBT event is generated when a single step occurs or when a debug trap, such as an event triggered by one of the hardware

debug registers, occurs.

Event	Code	Description
KDP_T_SSTEP	1	Single step or debug trap

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_TBT
Value	Not used.
OffV	Linear address in CS:(E)IP at time of event.
SegV	Slot number of thread.
MTE	MTE entry of executable running in process.
PID	PID of process that generated the event.
TID	TID of thread that generated the event.
DBit	Flags from CS selector.
Reg	Registers at time of event.
MemCache	Not used.

CVK_RET_BPT Event

Event	Code	Description
KDP_T_BREAKPOINT	3	Software breakpoint

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_BPT
Value	Not used.
OffV	Linear address in CS:(E)IP at time of event.
SegV	Slot number of thread.
MTE	MTE entry of executable running in process.
PID	PID of process that generated the event.
TID	TID of thread that generated the event.
DBit	Flags from CS selector.
Reg	Registers at time of event.
MemCache	Not used.

CVK_RET_NMI Event

Event	Code	Description
KDP_T_NMI	2	Non-maskable interrupt

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_NMI
Value	Not used.
OffV	Linear address in CS:(E)IP at time of event.
SegV	Slot number of thread.
MTE	MTE entry of executable running in process.
PID	PID of process that generated the event.
TID	TID of thread that generated the event.
DBit	Flags from CS selector.
Reg	Registers at time of event.
MemCache	Not used.

CVK_RET_SUC Event

Event	Code	Description
KDP_T_SUCCESS	100	When last CVK_CMD_RAW response is sent.

Cmd	CVK_RET_SUC

CVK_RET_ASYNC Event

Event	Code	Description
KDP_T_ASYNC_TRAP	101	KDP break received.

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_ASYNC
Value	Not used.

OffV	Linear address in CS:(E)IP at time of event.
SegV	Slot number of thread.
MTE	MTE entry of executable running in process.
PID	PID of process that generated the event.
TID	TID of thread that generated the event.
DBit	Flags from CS selector.
Reg	Registers at time of event.
MemCache	Not used.

CVK_RET_LIB and CVK_RET_KIL Events

Event	Code	Description
KDP_T_LINK	102	Module loaded
KDP_T_UNLINK	103	Module unloaded

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_LIB or CVK_RET_KIL
Value	MTE handle of module in question.
OffV	Not used.
SegV	Slot number of thread.
MTE	MTE entry of executable running in process.
PID	PID of process that generated the event.
TID	TID of thread that generated the event.
DBit	Not used.
UCHAR NAME[]	Null-terminated full pathname of module in question (immediately follows *DBit* and overlays *Reg* and *MemCache*)

CVK_RET_TNEW and CVK_RET_TEND Events

Event	Code	Description
KDP_T_TASK_CREATE	104	Task create
KDP_T_TASK_END	105	Task died

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_TNEW or CVK_RET_TEND
Value	Not used.
OffV	Not used.
SegV	Not used.
MTE	Not used.
PID	PID of process that was created or died.
TID	1 (Primary TID in process)
DBit	Not used.
Reg	Not used.
MemCache	Not used.

CVK_RET_PAGEIN Events

Event	Code	Description
KDP_T_PAGEIN	106	Discarded page reloaded

Fields returned in cvkcmd_s are as follows:

Cmd	CVK_RET_PAGEIN
Value	Not used.
OffV	Address of the page in question.
SegV	Not used.
MTE	MTE handle of the module that contains the page in question.
PID	Not used.
TID	Not used.
DBit	Not used.
Reg	Not used.
MemCache	Not used.

Kernel Debugger Packet Commands

The following table lists the kernel debugger packet commands:

Command	Code	Description	CVK_CMDSIZE_	CVK_RETSIZE_

CVK_CMD_RMEM	1	Read memory	18	20
CVK_CMD_RREG	3	Read registers	18	24 + sizeof(RegSa_ struc)
CVK_CMD_WMEM	4	Write memory	20	6
CVK_CMD_WREG	6	Write registers	20 + sizeof(RegSa_ struc)	2
CVK_CMD_RUN	7	Resume execution	6	0
CVK_CMD_KILL	8	Reboot victim machine	2	0
CVK_CMD_STEP	9	Single step	2	0
CVK_CMD_NUMTOBASE	13	Get object/segment information	14	14
CVK_CMD_LIBNAME	16	Get module information	6	6
CVK_CMD_RAW	20	Perform kernel debugger command	6	
CVK_CMD_DBIT	22	Get selector information	20	
CVK_CMD_RSTEP	23	Range step	10	0
CVK_CMD_SCANMTE	24	Scan module table	2	6
CVK_CMD_SCANTCB	25	Scan thread control blocks	6	10
CVK_CMD_SEL2LIN	26	Convert selector:offset to linear address.	18	6
CVK_CMD_LIN2SEL	27	Convert linear address to selector:offset.	18	12
CVK_CMD_OBJCOUNT	28	Get number of objects/segments in module	6	6
CVK_CMD_SCANOBJ	29	Scan object/segment table	14	10
CVK_CMD_SELINFO	30	Get selector information	18	20
CVK_CMD_RNPX	31	Read NPX state	18	128
CVK_CMD_WNPX	32	Write NPX state	128	60
CVK_CMD_ENA	33	Enable optional features	6	2
CVK_CMD_DIS	34	Disable optional features	6	2
CVK_CMD_PIREG	35	Register for PAGEIN notification	14	2
CVK_CMD_PIDRG	36	Deregister for PAGEIN notification	14	2

Each command is described below, along with the input parameters and the output values. Parameters are usually passed in a cvkcmd_s If a given field in the cvkcmd_s structure is not listed, its value is not used.

There are constants called CVK_CMDSIZE_xxx and CVK_RETSIZE_xxx which provide the size of the command and the size of the response from each command, where **xxx** is substituted with the last part of the command name. For CVK_CMD_PIDRG, the constants would be called CVK_CMDSIZE_PIDRG and CVK_RETSIZE_PIDRG. The values for these constants are provided in the last two rows of the table above.

Read Memory (CVK_CMD_RMEM)

Parameters:

Cmd CVK_CMD_RMEM

Value Number of bytes to read. (<= CVK_MEMCACHE_SIZE)

OffV Offset of data to read.

SegV Selector or segment index for data to read. 0 if *OffV* is a linear address, otherwise setting of PE in CR0 and VM in EFLAGS of interrupted thread determine whether *SegV* is selector or selector index.

TID Slot of thread in whose context address is to be used. 0 is used for regions in globally shared areas.

Returns:

Value Number of bytes actually read.

MemCache [*Value*] Data read.

Read Registers (CVK_CMD_RREG)

Parameters:

Cmd CVK_CMD_RREG

TID Slot of thread in whose registers are desired.

Returns:

Reg Registers for thread.

ULONG PC (immediately following *Reg*) Linear address corresponding to CS:(E)IP.

Write Memory (CVK_CMD_WMEM)

Parameters:

Cmd CVK_CMD_WMEM

Value Number of bytes to write. (<= CVK_MEMCACHE_SIZE)

OffV Offset of data to write.

SegV Selector or segment index for data to write. 0 if *OffV* is a linear address, otherwise setting of PE in CR0 and VM in EFLAGS of interrupted thread determine whether *SegV* is selector or selector index.

TID Slot of thread in whose context address is to be used. 0 is used for regions in globally shared areas.

MemCache [*Value*] Data to write.

Returns:

Value Number of bytes actually written.

Write Registers (CVK_CMD_WREG)

Parameters:

Cmd CVK_CMD_WREG

TID Slot of thread in whose registers are to be changed.

Reg Registers for thread. Only the following registers are affected: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, EIP, CS, DS, ES, FS, GS, SS, and EFLAGS.

Returns:

Cmd Return code

Resume Execution (CVK_CMD_RUN)

Parameters:

Cmd CVK_CMD_RUN

Value Indicates action kernel debugger should take after resuming execution.

-CVK_CMD_RUN

Kernel debugger is not to expect further commands or generate further events. All optional features enabled with the CVK_CMD_ENA command are disabled. In other words, this command detaches the debug engine from the kernel debugger.

Anything else

Kernel debugger sends notification when an event occurs.

Returns:

None. See description above in *Value* for action taken by system under debug.

Reboot Victim (CVK_CMD_KILL)

Parameters:

None.

Returns:

None.

Single Step (CVK_CMD_STEP)

Parameters:

None.

Returns:

None. System under debug sends event notification when an event occurs.

Get Object/Segment Information (CVK_CMD_NUMTOBASE)

Parameters:

Cmd	CVK_CMD_NUMTOBASE
Value	Object/segment index for segment about which information is desired.
MTE	MTE handle for module about which information is desired.

Returns:

Value	Flags from object/segment table entry for segment. (ote_flags or ste_flags)
OffV	Base address for object. (0 for segments.)
SegV	Selector for object/segment.

Get Module Information (CVK_CMD_LIBNAME)

Parameters:

Cmd	CVK_CMD_LIBNAME
Value	MTE handle for module about which information is desired.

Returns:

Value	Length of full pathname of module, including trailing null character. Returns 0 if SMTE or name is not resident.

UCHAR NAME[*Value*] Null-terminated full pathname of module (immediately follows *Value*)

Perform Kernel Debugger Command (CVK_CMD_RAW)

Parameters:

Cmd	CVK_CMD_RAW
OffV	Null-terminated text of command to perform (overlays *OffV*)

Returns:

Cmd	Indicates whether this is the last response.	
	CVK_CMD_RAW	More response packets coming.
	CVK_RET_SUC	Last response packet.
Value	Length of command response.	
OffV	Null-terminated response to command (overlays *OffV*)	

Note: This command may return multiple responses. The last event packet response will have *Cmd* set to CVK_RET_SUC instead of CVK_CMD_RAW.

Get Selector Information (CVK_CMD_DBIT)

Parameters:

Cmd CVK_CMD_DBIT

SegV Selector for which information is desired.

Returns:

DBit Flags from selector's descriptor.

Range Step (CVK_CMD_RSTEP)

Parameters:

Cmd CVK_CMD_RSTEP

OffV Offset at which to stop.

Returns:

None. System under debug steps until CS selector changes or until (E)IP becomes >= *OffV* value, then issues a single step event.

Scan Module Table (CVK_CMD_SCANMTE)

Parameters:

Cmd CVK_CMD_SCANMTE

Returns:

Cmd CVK_RET_LIB if information was returned, otherwise returns CVK_RET_SUC if end of module table reached.

Value MTE handle of module if *Cmd* = CVK_RET_LIB.

UCHAR NAME[*Value*] Null-terminated full pathname of module (immediately follows *Value*)

Not present if SMTE or name is not paged in. Packet length is CVK_RETSIZE_SCANMTE in this case.

The module table scan ignores resource modules, which contain only resource objects and segments, and forwarder modules, which contain no objects or segments.

Scan Thread Control Blocks (CVK_CMD_SCANTCB)

Parameters:

Cmd CVK_CMD_SCANTCB

Value Slot number of thread control block to start with. Specify 0 to start at first occupied slot. Specify -1 to start at currently scheduled slot.

Returns:

Value Slot number of next occupied thread control block. 0 if no more TCBs.

OffV Number of information blocks returned in current response.

```
struct {
  USHORT slot;   /* Slot index for thread */
  USHORT PID;    /* PID that contains thread */
  USHORT TID;    /* Thread ordinal */
  USHORT MTE;    /* MTE handle for module running in process
                    that contains thread */
  USHORT flags;  /* Flags for thread
                    Bit 0 (low-order bit)  If ON, indicates
                         thread was running when system stopped.
                    Bits 1-15 are unused  */
  }  TIBInfo
```

Information returned immediately following *OffV*.

Convert Selector:Offset to Linear Address (CVK_CMD_SEL2LIN)

Parameters:

Cmd CVK_CMD_SEL2LIN

OffV Offset

SegV Selector

TID Thread that linear address should be in context of.

Returns:

Value Linear address

Convert Linear Address to Selector:Offset (CVK_CMD_LIN2SEL)

Parameters:

Cmd CVK_CMD_LIN2SEL

Value Linear address

TID Slot of thread that desired linear address is to be in context of. (Currently ignored. Linear address must be in global region and mapped by interrupted thread's CS, DS, ES, or SS.)

Returns:

OffV Offset

SegV Selector

Get Number of Objects/Segments in Module (CVK_CMD_OBJCOU

Parameters:

Cmd CVK_CMD_OBJCOUNT

Value MTE handle for module about which information is desired.

Returns:

Value Number of objects/segments in module. 0 returned if the module is a forwarder module, which contains no objects or segments, or a resource module, which contains only resource objects and segments.

Scan Object/Segment Table (CVK_CMD_SCANOBJ)

Parameters:

Cmd CVK_CMD_SCANOBJ

Value Object/segment table tindex of entry to start with. Specify 0 to start at the beginning.

MTE MTE handle for module about which information is desired.

Returns:

Value Object/segment table index of next entry. 0 is returned if no more entries.

OffV Number of information blocks returned in current response.

```
struct {
   ULONG  base;   /* Base address of object/segment */
   ULONG  size;   /* Size in bytes of object/segment in memory */
   ULONG  flags;  /* Flags for object/segment:
                     bit 0 (low-order bit)
                        ON  - 16-bit segment
                        OFF - 32-bit segment
                     bit 1
                        ON  - data
                        OFF - code
                     bits 2-31 are unused */
} ObjInfo;
```

Information block returned immediately follows *OffV*

Get Selector Information (CVK_CMD_SELINFO)

Parameters:

Cmd CVK_CMD_SELINFO

SegV Selector

TID Slot of thread desired selector is to be in context of.

Returns:

Value Limit of segment in bytes.

OffV Base address of segment.

DBit Flags from descriptor.

Read NPX Information (CVK_CMD_RNPX)

Parameters:

Cmd CVK_CMD_RNPX

TID Slot of thread whose NPX state is desired.

Returns:

Cmd Return code.

0	NPX state read and returned
1	NPX is being emulated.
2	Invalid slot.
3	Thread has no NPX state.
4	State swapped out.

Reg Image of NPX state as saved by FSAVE instruction.

Write NPX Information (CVK_CMD_WNPX)

Parameters:

Cmd CVK_CMD_WNPX

TID Slot of thread whose NPX state is to be set.

Reg Image of NPX state (restored by FRESTOR instruction)

Returns:

Cmd Return code.

0	NPX state written successfully.
1	NPX is being emulated.
2	Invalid slot.
3	Thread has no NPX state.
4	State swapped out.

Enable Optional Features (CVK_CMD_ENA)

Parameters:

Cmd CVK_CMD_ENA

Value Index of feature to enable.

0	CVK_RET_TNEW and CVK_RET_TEND notifications.

Returns:

Cmd Return code.

Note: All optional features are disabled when a CVK_CMD_RUN command is issued with a *Value* of -CVK_CMD_RUN.

Disable Optional Features (CVK_CMD_DIS)

Parameters:

Cmd	CVK_CMD_DIS	
Value	Index of feature to disable.	
	0	CVK_RET_TNEW and CVK_RET_TEND notifications.

Returns:

Cmd	Return code.

Note: All optional features are disabled when a CVK_CMD_RUN command is issued with a *Value* of -CVK_CMD_RUN.

Register for PAGEIN Notification (CVK_CMD_PIREG)

Parameters:

Cmd	CVK_CMD_PIREG
OffV	Address of page.
MTE	MTE handle of module that contains page.

Returns:

Cmd	Return code.

Deregister for PAGEIN Notification (CVK_CMD_PIDRG)

Parameters:

Cmd	CVK_CMD_PIDRG
OffV	Address of page.
MTE	MTE handle of module that contains page.

Returns:

Cmd	Return code.

Device I/O

Devices used with computers include the keyboard, video display, mouse, floppy and fixed disk drives, and external systems, such as modems and printers. This chapter describes the OS/2 functions used to access and control such devices.

The following topics are related to the information in this chapter:

- File systems
- File names
- File management
- Semaphores

About Device I/O

OS/2 uses devices to communicate with the real world. A device is a piece of hardware used for input and output. The keyboard and screen are devices, as are serial and parallel ports. The computer's speaker, which can be made to beep using DosBeep, is a device.

Accessing Devices

OS/2 applications usually communicate with devices through OS/2. Some devices, like the screen, have their own set of supporting functions. Most other devices can be accessed by using the standard OS/2 file system functions- DosOpen, DosRead, DosWrite, and DosClose. Using the file system functions, an application can open and access the device just as it would a disk file. Using the file system also enables applications to redirect the device's I/O stream.

Sometimes however, these higher-level approaches do not suffice. For these situations, OS/2 provides several functions that interface with devices at a lower level. DosDevConfig is used to retrieve information about the devices available. DosPhysicalDisk can be used to retrieve information about a partitionable hard disk. DosDevIOCtl is used to send device-specific commands directly to a particular device driver.

Device Names

To open a device using DosOpen, the application must supply the reserved name for that device. For example, to open the console (both keyboard and screen), you must specify the name CON.

The following table shows some of the common reserved device names:

Common Device Names

Device Name	Description
CON	The system console. This device consists of both the keyboard and the screen. You can open CON for reading (from the keyboard), writing (to the screen), or both.
COM1	Serial port 1. You can open this device for reading, writing, or both. Other serial ports will have names in ascending sequence-COM2, COM3, and so on.
PRN	The default printer port. This device corresponds to one of the system's parallel ports, usually LPT1. You can open it for writing but not for reading.
LPT1	Parallel port 1. You can open this device for writing but not for reading. Other parallel ports will have names in ascending sequence-LPT2, LPT3, and so on.
NUL	The null device. This device provides a method of

```
              discarding output. If you open this device for
              writing, any data written to the file is
              discarded. If you open the device for reading, any
              attempt to read from the file returns an
              end-of-file mark.

SCREEN$       The screen. This device can be written to but not
              read from. Writing to the screen is similar to
              writing to the system console. Bytes are displayed
              as characters (unless the ANSI screen driver is
              loaded and the character represents an ANSI escape
              sequence).

KBD$          The keyboard. This device can be read from but not
              written to. Reading from the keyboard is similar
              to reading from the system console.
```

After an application uses a device, it should close it by using DosClose.

Device Drivers

OS/2 communicates with devices through special programs called *device drivers* . A device driver acts as an interface between OS/2, together with its applications, and a physical device such as the keyboard, mouse, or printer. The device driver sends data to and receives data from a device, resolving device-independent requests from applications with the device-specific attributes of the device.

The primary method of communication between OS/2 and a device driver is request packets. OS/2 receives I/O requests from applications and sends data in the form of request packets to the device driver. The device driver communicates with the device either directly or through the BIOS and ABIOS interfaces. (Applications can communicate with device drivers also, by using DosDevIOCtl. See IOCtl Interface)

Devices work differently depending on the device driver installed. For example, if an application writes to the system console, each byte is interpreted as a character and is displayed on the screen. If, however, the ANSI display driver is loaded, some byte sequences direct the system to carry out certain actions on the screen, such as moving the cursor or clearing the screen. These byte sequences are called ANSI *escape sequences* .

Some devices are available to applications only if the appropriate device driver is installed. For example, an application cannot open a serial port unless a communications device driver, such as COM.SYS, has been loaded by using a DEVICE= command in CONFIG.SYS.

IOCtl Interface

Many devices have more than one operating mode. For example, a serial port typically can operate at a variety of bit rates (sometimes called baud rates). Because the modes are unique to each device, OS/2 does not include specific functions to set or retrieve these modes. Instead OS/2 provides an I/O Control (IOCtl) interface to enable applications to control devices by communicating directly with the device driver.

The IOCtl interface is a method that an application or subsystem can use to send device-specific control commands to a device driver. The IOCtl interface function for OS/2 applications is DosDevIOCtl.

DosDevIOCtl provides a generic, expandable IOCtl facility. Applications send commands and data to the device driver with DosDevIOCtl. The OS/2 kernel reformats the generic IOCtl packets into request packets then calls the device driver. The device driver then carries out the specified action. IOCtl commands can be sent to both block and character device drivers.

Before using DosDevIOCtl, the application or subsystem must first obtain the device handle by calling DosOpen for the device name. The opened device handle is used to specify the device the command is to go to.

Refer to the *Control Program Programming Reference* for details of DosDevIOCtl.

IOCtl Commands

DosDevIOCtl has many subfunctions. These are called generic IOCtl commands and typically are used to retrieve data from a device driver that is not available through standard OS/2 functions. For example, an application can use these functions to set the bit rate of a serial port or read input directly from a sector on a disk.

Category and Function Codes

Each IOCtl function has a category and a function code. The category defines the type of device to be accessed. OS/2 has several predefined categories. In general, all codes in the range 0x0000 through 0x007F are reserved for predefined categories. A device driver can use additional categories, but they must be explicitly defined by the device and be in the range 0x0080 through 0x00FF.

In each category, a function code defines the action to carry out, such as reading from or writing to the device and retrieving or setting the device modes. The number and meaning of each function code depend on the device driver and the specified category.

Parameter and Data Packets

DosDevIOCtl uses a parameter packet and a data packet to pass information to and from the device driver. The packets can vary in format and length, depending on the IOCtl function. Simple functions might use only a single variable, while more complex functions might require a more complex data structure for the parameter packet, the data packet, or both.

Using the File System to Access Devices

An application can use the OS/2 file system functions-DosOpen, DosRead, DosWrite, and DosClose-with the standard (predefined) devices. The application simply specifies the name of the device in the call to DosOpen, then uses the returned handle to read from and write to the device. When the application has finished using the device, the application should close the device handle by using DosClose.

The following code fragment shows how an application can open the COM1 device (serial port 1) and write the contents of a disk file to the communications port:

```
#define INCL_DOSFILEMGR    /* File System values */
#define INCL_DOSDEVIOCTL   /* DosDevIOCtl values */
#include <os2.h>

BYTE abBuf[512];
HFILE hfCom, hfFile;
ULONG ulAction, cbRead, cbWritten;

DosOpen("COM1",
        &hfCom,
        &ulAction,
        0,
        FILE_NORMAL,
        FILE_OPEN,
        OPEN_ACCESS_READWRITE |
        OPEN_SHARE_DENYNONE,
        (PEAOP2) NULL);

DosOpen("testfile",
        &hfFile,
        &ulAction,
        0,
        FILE_NORMAL,
        FILE_OPEN,
        OPEN_ACCESS_READONLY |
        OPEN_SHARE_DENYWRITE,
        (PEAOP2) NULL);

do {
    DosRead(hfFile,
            abBuf,
            sizeof(abBuf),
            &cbRead);
```

```
        DosWrite(hfCom,
                 abBuf,
                 cbRead,
                 &cbWritten);

    } while(cbRead);

    DosClose(hfCom);

    DosClose(hfFile);
```

Note: In this example code fragment and the ones that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Using IOCtl Functions to Access Devices

Many OS/2 functions communicate with devices. Usually, this communication is transparent to the application (the application has no knowledge of how the communication actually occurs). At times, however, an application requires more direct access to a device. To accommodate this need, OS/2 furnishes DosDevIOCtl. Applications can use DosDevIOCtl to send commands and data to a device driver; the device driver interprets these commands and sends the appropriate instructions to the physical device.

As an example, some devices have several operating modes. A communications port can operate at one of a number of bit rates and have several data-word formats. The actual commands to set these parameters might vary, depending on the communications hardware.

Named constants have been defined for the categories, functions, and commands that are passed to a device driver, to make it easier for application programmers to use DosDevIOCtl. These named constants are defined in the file BSEDEV.H. This file must be included in your application when you use the constants. This file also contains data structure definitions for the parameter and data packets commonly used with DosDevIOCtl. The following examples use the communications port to demonstrate how DosDevIOCtl works.

Setting Communications-Port Parameters

You can use DosDevIOCtl to control the data parameters (bit rate, stop bits, parity, and data bits) of a communications port and to get the status of the COM port. The IOCTL_ASYNC category is used for communications-port control. The ASYNC_SETBAUDRATE function sets the COM port transmission rate. The ASYNC_GETCOMMSTATUS returns the COM port status-byte.

Setting the Data Rate

The ASYNC_SETBAUDRATE function sets the bit rate of a communications port.

The following code fragment sets the bit rate of COM1 to 9600 bits per second:

```
    #define INCL_DOSFILEMGR    /* File System values */
    #define INCL_DOSDEVIOCTL   /* DosDevIOCtl values */
    #include <os2.h>

    HFILE   hf;                /* File handle for the device           */
    USHORT  usBPS = 9600;      /* Bit rate to set the COM port to      */
    ULONG   ulParmLen = 2;     /* Maximum size of the parameter packet */
    ULONG   ulAction;          /* Action taken by DosOpen              */
    APIRET  ulrc;              /* Return code                          */

    ulrc = DosOpen("COM1",
                   &hf,
                   &ulAction,
                   0,
                   FILE_NORMAL,
```

```
                FILE_OPEN,
                OPEN_ACCESS_READWRITE |
                OPEN_SHARE_DENYNONE,
                (PEAOP2) NULL);

    ulrc = DosDevIOCtl(hf,                /* Device handle                  */
                       IOCTL_ASYNC,       /* Serial-device control          */
                       ASYNC_SETBAUDRATE, /* Sets bit rate                  */
                       (PULONG) &usBPS,   /* Points at bit rate             */
                       sizeof(usBPS),     /* Maximum size of parameter list */
                       &ulParmLen,        /* Size of parameter packet       */
                       NULL,              /* No data packet                 */
                       0,                 /* Maximum size of data packet    */
                       NULL);             /* Size of data packet            */
    .
    .   /* Use the COM port here. */
    .

    ulrc = DosClose(hf);
```

Getting the COM Port Transmission Status

The ASYNC_GETCOMMSTATUS function get the transmission status of the specified COM port. This function has no parameter packet.

The following code fragment uses the ASYNC_GETCOMMSTATUS function to get the transmission status of COM1:

```
    #define INCL_DOSFILEMGR    /* File System values */
    #define INCL_DOSDEVIOCTL   /* DosDevIOCtl values */
    #include <os2.h>

    HFILE   hf;           /* File handle for the device         */
    UCHAR   ucStatus;     /* The COM port status byte           */
    ULONG   ulStatusLen;  /* Length of status (the data packet) */
    ULONG   ulAction;     /* Action taken by DosOpen            */
    APIRET  rc;           /* Return code                        */

    rc = DosOpen("COM1", &hf, &ulAction, 0,
                 FILE_NORMAL, FILE_OPEN,
                 OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
                 (PEAOP2) NULL);

    rc = DosDevIOCtl(hf,                  /* Device handle                    */
                     IOCTL_ASYNC,         /* Serial-device control            */
                     ASYNC_GETCOMMSTATUS, /* Get the COM status byte          */
                     NULL,                /* No parameter packet              */
                     0,                   /* Maximum size of parameter packet */
                     NULL,                /* Length of parameter packet       */
                     (PULONG) &ucStatus,  /* Data packet                      */
                     sizeof(ucStatus),    /* Maximum size of data packet      */
                     &ulStatusLen);       /* Length of data packet            */
    .
    .   /* Use the COM port here. */
    .

    rc = DosClose(hf);
```

Dynamic Linking

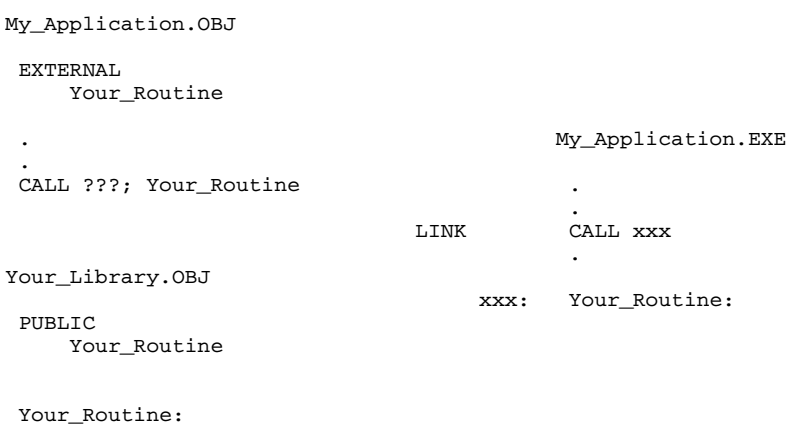
This chapter describes:

- Static versus dynamic linking
- Dynamic link library (DLL) data

- DLL initialization and termination
- Building a DLL
- Use of protected memory by DLLs

Static Versus Dynamic Linking

Most programmers are familiar with static linking; an application calls a routine or procedure whose code is not found in the application's source file. The routine is external to your source file and is declared as such. When the source file is compiled, the compiler places an external reference for the routine in the application's object (.OBJ) file. To create the executable file (.EXE) for the application, the application's object file is linked with an object file that contains the code for the routine. The result is an EXE file that contains the application code, as well as a copy of the code for the routine. The following figure illustrates the process of building a statically linked application.



Static Linking

When OS/2 loads a statically linked program, all the code and data are contained in a single EXE file and the system can load it into memory all at once.

The advantages and disadvantages of static linking are summarized in the following table.

Advantages	Disadvantages
Compile in pieces	External routines built into EXE (making EXEs larger)
Can create libraries of routines that can be linked with applications.	EXE cannot be changed without relinking.
	External routines cannot be shared (duplicate copies of libraries).

Dynamic linking permits several applications to use a single copy of an executable module. The executable module is completely separate from the applications that use it. Several functions can be built into a DLL, and applications can use these functions as though they were part of the application's executable code. You can change the dynamically-linked functions without recompiling or relinking the application.

The advantages of dynamic linking are:

Reduced memory requirements

Many applications can use a single DLL simultaneously. Since only one copy of the DLL is in memory, this saves memory space and, in general, the code is discardable.

Simplified application modification

An application can be enhanced or modified simply by changing a DLL. For example, if an application uses a DLL to support video

output, several displays can be supported by different DLLs. The application can use the DLL that supports the appropriate display.

Flexible software support

DLLs can be used for after-market support. In the display-driver example, a new DLL can be provided to support a display that was not available when the application was shipped. Similarly, a database application can support a new data-file format with a new DLL.

Transparent migration of function

The DLL functions can be used by applications without any understanding of how the functions actually do their work. Future changes to the DLL are transparent to the application, as long as the input and output parameters remain the same.

Multiple programming language support

A function in a DLL can be used by an application written in any programming language, as long as the application understands the function's calling convention.

Application-controlled memory usage

Applications can make decisions about which DLL routines they want to load into memory and use. If a DLL is not used, it does not have to be loaded.

DLLs can be used to implement subroutine packages, subsystems, and interfaces to other processes. In OS/2:

- Some DLLs are interfaces to the kernel.

 The worker routines for the OS/2 API reside in the OS/2 kernel. Applications, which run at privilege level 3, usually can make direct calls to the kernel, which runs at privilege level 0. Some calls, however, must be linked through a DLL. For example, an application that calls DosOpen is linked to the DLL DOSCALL1.DDL. This library contains the definitions for some system functions. When a system function is called, OS/2 makes the call to the kernel on behalf of the application.

- Some DLLs are interfaces to devices.

 DLL subsystems can virtualize devices and manage the device for its clients.

- Some DLLs provide an open system architecture for software.

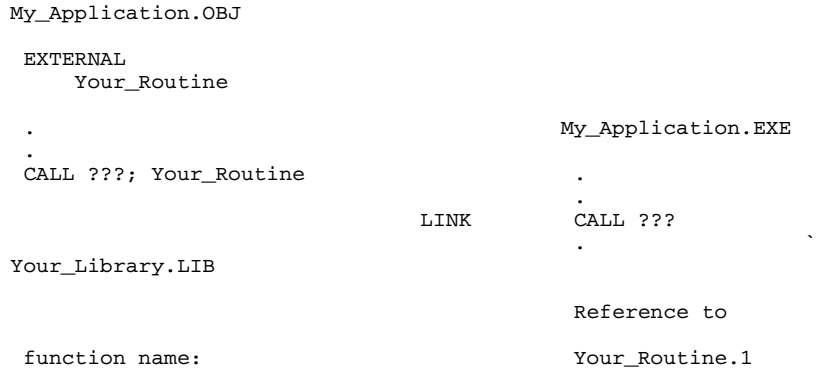
 Add-ons to OS/2 can be provided easily and by anyone.

OS/2 provides two varieties of dynamic linking: *load-time* and *run-time* . In load-time dynamic linking, references are resolved when an application is loaded. In run-time dynamic linking, references are resolved when the application runs.

Load-Time Dynamic Linking

Load-time dynamic linking is similar to static linking in that an application can call a routine that is not found in the application's source file. In load-time dynamic linking, however, an application is linked with a library file that contains a record that describes where the routine can be found instead of with a file that contains the code for the routine. The resulting application executable file contains this record and not a copy of the routine's code. The following figure illustrates the process of building a load-time dynamically linked application.

In the example in the following figure, the LIB file contains a record that describes where the code for a set of functions can be found. In this case, the code for the function Your_Routine can be found in the file, or module, Your_Routines.DLL under the entry point name Your_Routine. (The entry point name does not have to match the function name.) The function code also can be referenced by its ordinal value.

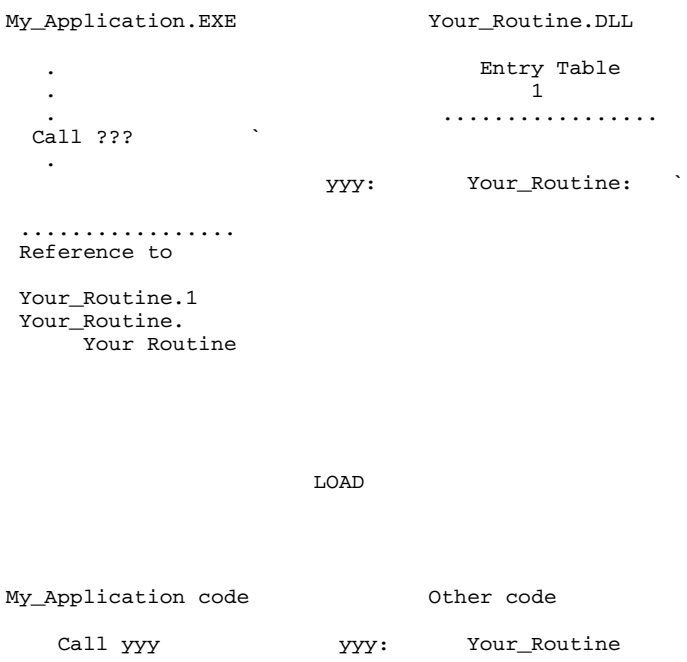


```
          Your_Routine                          Your_Routine.
                                                    Your_Routine
    module name:
          Your_Routines

    entry point
          ordinal value: 1
          entry point name
              Your_Routine
```

Dynamic Linking

When the application is loaded, the system resolves the dynamic-link references, as shown in the following figure.



Resolving Dynamic Link References

If a program contains dynamically linked references, the system must process the information in the references. If the DLL already is in memory, the system adds information to the executable code so that the code can use the DLL functions. If the required DLLs are not already in memory, OS/2 searches the path specified by the LIBPATH environment variable. If the system cannot find the DLLs, it stops loading the application and reports the error. If the system finds the DLLs, it loads them. When DLLs are loaded into memory, OS/2 notifies the application where the DLL functions can be found.

When a DLL is loaded into memory is determined by how the DLL was built. A DLL is built like an application, using a module-definition (.DEF) file. The CODE statement in a DEF file describes the attributes of application or DLL code. The *load* option for the CODE statement determines when application or DLL code is loaded:

PRELOAD	Code is loaded as soon as a process accesses the DLL. This leads to increased performance (in terms of accessing the DLL functions) while decreasing available memory. This option might be preferable if only one program is running.
LOADONCALL	Code is loaded when needed. This is the recommended choice and the default.

Run-Time Dynamic Linking

When an application contains a reference to a DLL, the DLL is loaded into memory when the application is loaded (unless the DLL already is

in memory). If the application uses functions in several DLLs, all of those DLLs are loaded into memory. Some applications might use functions in several DLLs but use only a few of them at any one time. For example, an application that supports many different printers or plotters might use functions in many DLLs but need only one of them at a time, depending on the printer or plotter the application is supporting. If the user switches to a different printer or plotter, another DLL will be used, but the first will remain in memory. Loading DLLs this way can be very wasteful.

To avoid this problem, applications can take advantage of run-time dynamic linking and load and unload DLLs as they are required. The process of building a run-time dynamically linked application is similar to the process of building a load-time dynamically linked application. However, the EXE file for a run-time dynamically linked application does not contain a record describing where the external routines can be found. Instead, the application explicitly tells OS/2 when to load and free the dynamic link module.

Applications load and unload DLLs and call functions whose code resides in those DLLs by:

1. Calling DosLoadModule to get a handle to the DLL module.

 This function also loads the DLL into memory or attaches to it, if it already is loaded.

2. Calling DosQueryProcAddr to get a pointer to a function within the DLL.

3. Calling the function indirectly through the pointer returned by DosQueryProcAddr.

4. Calling DosFreeModule to free the handle to the DLL module.

 When all handles to the DLL module have been freed, the DLL is unloaded from memory.

An application also can request information about a DLL from the system. The application can use the DosQueryModuleHandle function to determine whether a DLL has been loaded into memory already. The DosQueryModuleName function returns full path information for the DLL file.

Following are the advantages of run-time dynamic linking:

Memory is consumed as needed.

DLLs can be loaded and unloaded as they are used. Unused DLLs do not have to be loaded into memory, and memory can be released when the application has finished using the DLL.

Applications can recover from DLL NOT FOUND.

Applications can make decisions. If a load-time DLL cannot be found, the application terminates immediately. If a run-time DLL cannot be found, the application receives an error value from the DosLoadModule function, and it can use another DLL or specify a full path for the DLL. If a function is not available, the DosQueryProcAddr function returns an error value, and the application can take appropriate action.

DLL and function names can be variable.

An application programmer does not have to know the names of the DLLs the application will be using or the names of the functions within the DLL. The application can read the names of the DLL or the functions from a configuration file or obtain the names from user-supplied input.

DLLs can be anywhere.

The application can specify the full path for the DLL. Load-time DLLs must be in a directory in the path specified by the LIBPATH environment variable, but run-time DLLs can be in other directories.

DLL Data

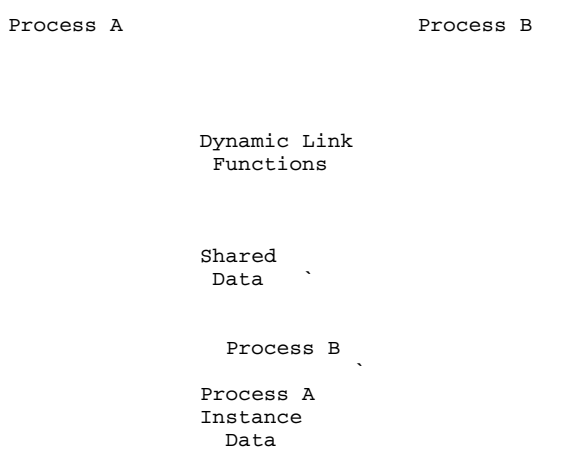
Most DLLs will contain some data. Whereas DLL code is shared by all processes that use it, DLL data can be shared or not shared by all processes that use it. Data that is specific to each process that uses the DLL (that is, to each instance of the DLL) is called *instance data* . Data that is shared by all processes that use the DLL is called *shared* or *global data* .

The first time a process references a DLL (and it is loaded or its usage count is incremented because it is already loaded), a separate copy of the DLL's instance data is created. Modifications to the instance data for one process do not affect the instance data for any other process. The system, however, maintains only one copy of a DLL's shared data for all processes that use the DLL. Every process that uses the DLL can access the same data. If one process modifies the data (increments a count, for example), the data will be modified for all processes that use the DLL.

Because changes to shared DLL data by one process are visible to the DLL code when called by another process, shared data provides DLLs with a mechanism for tracking processes that use it. This is crucial to subsystems, which are DLLs that manage resources (for example, devices, queues, and so forth).

There usually is only one data and one code object, or segment, defined in a C source file. This means that a DLL that has instance and shared data is built from more than one C source file, with a default automatic data segment and with named data segments. How data is defined is determined by how the DLL is built. A DLL is built like an application, using a DEF file. The DATA statement in a DEF file describes the attributes of application or DLL data. Following is a list of the available options for the DATA statement:

Options	Result
MULTIPLE	OS/2 creates a unique copy of the automatic data segment for each process that uses the DLL. Modifications made by one process do not affect any other process. This is the default.
SINGLE	OS/2 creates only one automatic data segment for all processes that use the DLL. If one process modifies the data, the data will be modified for all processes that use the DLL.
READWRITE	Enables you to read from or write to the automatic data segment. This is the default.
READONLY	Enables you to read only from the automatic data segment.
LOADONCALL	The automatic data segment is loaded into memory as needed. This is the default.
PRELOAD	The automatic data segment is loaded as soon as a process accesses a DLL.



DLL Data

You also can create a DLL with both shared and instance data. For more information, see Using Shared and Instance Data

DLL Initialization and Termination

A DLL is initialized and terminated by the default _DLL_InitTerm function. When a process gains access to the DLL, this function initializes the necessary environment for the DLL, including storage, semaphores, and variables. When the process frees its access to the DLL, the _DLL_InitTerm function terminates the DLL environment created for that process. The _DLL_InitTerm function is called automatically when you link to the DLL.

The _DLL_InitTerm function can be executed once when the DLL is first loaded into memory, or it can be executed each time a new process first accesses the DLL. The LIBRARY statement in the module-definition file is used to specify when the _DLL_InitTerm function is to be executed. Following is a list of of the available options for the LIBRARY statement.

Options	Result
INITINSTANCE	The _DLL_InitTerm function is called the first time the DLL is loaded for each process that accesses the DLL.
INITGLOBAL	The _DLL_InitTerm function is called only the very first time the DLL is loaded. This is the default.
TERMINSTANCE	The _DLL_InitTerm function is called the last time the DLL is freed for each process that accesses the DLL.

As an example, the following statement, identifies the executable file as a DLL and specifies that SAMPLE03 is the name of the DLL:

```
LIBRARY SAMPLE03 INITINSTANCE TERMINSTANCE
```

It also specifies that the _DLL_InitTerm function is to be called the first time the DLL is loaded for each process that calls the DLL and the last time the DLL is freed for each process that calls the DLL.

When OS/2 starts executing a DLL, it sets the CPU registers to known values, but only for 16-bit DLLs. All 32-bit DLLs are called with a stack frame, like all other API calls.

Initialization/termination functions can be written in a high-level language. For more information on writing your own initialization/termination function, see Creating an Initialization/Termination Function.

Building DLLs

Building a DLL is not very different from building a conventional static library. The following sections show how you can use OS/2 tools and functions to create, manage, and use DLLs.

External Function References

When you compile or assemble an application, the compiler or assembler generates an object module for the code in the application. If you use any functions that are external to your application (their code is in another object module), the compiler or assembler adds an external function reference to your application's object module.

The Linker resolves these external references. If the Linker finds the external function in a library called an import library or in an IMPORTS statement in the application's module-definition file, the code for the external function is in a DLL. To resolve external references to DLLs, the Linker simply adds information to the executable file that tells the loader where to find the DLL code when the application is loaded.

Module-Definition Files

The module-definition file is an important tool for building DLLs. This file contains information that tells OS/2 the name of the DLL, when to load the DLL, how to manage memory for the DLL, and when to initialize the DLL.

When you create a DLL, the module-definition file must contain a list of all the functions in the DLL that can be called by an application (or by another DLL). You specify these external functions by using an EXPORTS statement in the module-definition file.

You also must tell the Linker where to find the external functions in your application. If the functions are in a DLL, you can use an IMPORTS statement in the module-definition file for the application to tell the Linker where to find the DLL functions. You also can use an import library to tell the Linker where to find your DLL functions.

Import Libraries

A conventional library contains object modules for a number of functions. The library is a convenient way to manage a large number of modules and use them in your executable code by linking to the library. The Linker uses the external references in your object module to determine which modules must be pulled out of the library.

An import library does not contain any object modules. Instead, the import library contains information that tells the Linker what DLLs are used by your application and the location of the functions your application uses within each DLL.

Like a conventional library, an import library primarily is a convenience. Instead of specifying all the functions your application imports in its

module-definition file, you can link with the import library and let the Linker resolve the external references in your object module.

You use import libraries every time you compile and link a program that uses the OS/2 API. All the OS/2 functions are implemented in DLLs, and OS2386.LIB is an import library that tells the Linker where to find each OS/2 function.

For more information about module-definition files and import libraries, see the online Tools Reference in the OS/2 Warp Developer's Toolkit.

Creating a Simple DLL

DLLs typically are used to provide common functions that can be used by a number of applications. The following figure shows a C source file, MYPUTS.C, for a DLL that contains a simple string-printing function.

```
#include <os2.h>

#define HF_STDOUT 1  /* standard-output handle               */

VOID EXPENTRY myPuts(PSZ pszMsg)
{
    ULONG ulWritten;

    while(*pszMsg) {
        DosWrite(HF_STDOUT,
                 pszMsg,
                 1,
                 &ulWritten);

        pszMsg++;
    }

}
```

The following figure shows the module-definition file, MYPUTS.DEF, for this DLL.

```
    LIBRARY myputs
    DATA SINGLE SHARED
    EXPORTS
        myPuts
```

The LIBRARY statement names the DLL (MYPUTS.DLL). The DATA statement tells the system that this DLL will share all data with each process that uses the DLL. The EXPORTS statement indicates that the function myPuts can be used by applications and DLLs.

The DLL is compiled and linked like any application. You can use IBM's ICC and LINK386, as shown below, to create MYPUTS.DLL.

```
    icc  /C+ /Ge- myputs.c

    link386 /noi myputs, , nul, OS2386, myputs
```

When the DLL has been created, you must copy it to one of the directories indicated by the LIBPATH environment variable in your CONFIG.SYS file.

Importing DLL Functions

After you create a DLL, you can use it in an application. The following figure shows a C source file, USEPUTS.C, that uses the myPuts function contained in the DLL MYPUTS.DLL.

```
    #include <os2.h>

    VOID EXPENTRY myPuts(PSZ);

    VOID main(VOID)
    {
```

```
    myPuts("Testing, 1,2,3");
}
```

The module-definition file for USEPUTS.C tells OS/2 where to find the myPuts function. This module-definition file (USEPUTS.DEF) contains the information shown in the following figure.

```
NAME useputs
IMPORTS
    myputs.myPuts
```

The module-definition file tells OS/2 that USEPUTS imports the myPuts function from MYPUTS.DLL. USEPUTS.C is compiled and linked as shown below.

```
icc  /C+ useputs.c

link386 /noi useputs, , nul, , useputs
```

Using an Import Library

You also can create an import library for your DLL. If you do this, you can link applications with your DLL without explicitly declaring the imports for each application. OS/2 uses this technique for the application programming interface (API). When you link your applications with OS2386.LIB, you are using an import library.

To create the import library STRINGS.LIB from MYPUTS.DLL, you use the Import Library Manager (IMPLIB), as shown below.

```
implib strings.lib myputs.def
```

You then can link your applications with STRINGS.LIB to resolve references to the myPuts function, as shown below.

```
link386 /noi useputs, , nul, strings;
```

A module-definition file for USEPUTS.C is optional in this example because we are linking with an import library.

Using Shared and Instance Data

When you create a DLL, you can use the DATA statement in the module-definition file to define the default attributes for data segments within the DLL. The default condition is for the DLL to have a unique copy of the automatic data segment for each process. You can specify DATA MULTIPLE READWRITE in the module-definition file to cause OS/2 to create a separate copy of all the DLL data for each process that uses the DLL (instance data). Modifications made by one process do not affect other processes.

You also can specify different attributes for different sets of data by using the #pragma data_seg and #pragma alloc_text directives to define your own data and code segments. You can list the segments in the module-definition file under the heading SEGMENTS and specify attributes for each.

```
SEGMENTS
   mydata SINGLE READONLY
   mycode PRELOAD
```

Any segments that you do not specify under SEGMENTS are given the attributes specified by the DATA or CODE statement, depending on the type of segment.

Creating an Initialization/Termination Function

It might be necessary for a DLL to perform some tasks before an application accesses a DLL or after an application finishes accessing a DLL. For example, the library might need to allocate a heap or open a device prior to using a DLL, or deallocate a heap or close a device after using a DLL. You can handle these tasks in an initialization/termination function. The initialization/termination function can be called to perform initialization tasks when the DLL is first loaded or each time a new process accesses the DLL, depending on the LIBRARY statement in the module-definition file. If you specify INITGLOBAL in the LIBRARY statement, the initialization/termination function is called only once, when the DLL is first loaded into memory. This is the default setting. If you specify INITINSTANCE, the library function is called each time the DLL is accessed by a new process. In the same way, the initialization/termination function can be called on to perform termination tasks. If you specify TERMINSTANCE, the library function is called each time the DLL is freed for each process that accesses the DLL.

When a thread calls DosLoadModule to load a DLL, the initialization routines of the loaded DLL (and the initialization routines of the DLLs that it loads) will run on the thread that called DosLoadModule. This initialization will complete before DosLoadModule returns.

The prototype for the _DLL_InitTerm function is:

```
unsigned long _DLL_InitTerm (unsigned long modhandle, unsigned long flag);
```

If the value of the *flag* parameter is 0, the DLL environment is initialized. If the value of the *flag* parameter is 1, the DLL environment is ended. The *modhandle* parameter is the module handle assigned by OS/2 for this DLL. The module handle can be used as a parameter to various OS/2 API calls. For example, DosQueryModuleName can be used to return the fully-qualified path name of the DLL, which tells you where the DLL was loaded from.

The return code from _DLL_InitTerm tells the loader whether the initialization or termination was performed successfully. If the call was successful, _DLL_InitTerm returns a nonzero value. A return code of 0 indicates that the function failed. If a failure is indicated, the loader will not load the program that is accessing the DLL.

Before you can call any C library functions, you must first initialize the C run-time environment. To initialize the environment, use the function _CRT_init. The prototype for this function is:

```
int _CRT_init (void);
```

If the run-time environment is successfully initialized, _CRT_init returns 0. A return code of -1 indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of stderr.

To properly terminate the C run-time environment, use the function, _CRT_term. The prototype for this function is:

```
void _CRT_term (unsigned long)
```

Because _DLL_InitTerm is called by OS/2, it must be compiled using the system linkage. In the IBM C Set/2 compiler, the following #pragma directive is used to specify the system linkage:

```
#pragma linkage (_DLL_InitTerm, system)
```

The initialization/termination function must have a specific entry point. You cannot create a function with a specific entry point in the C programming language, so the initialization function must be written in assembly language. However, you can write a very simple initialization function in assembly language and have it immediately jump to a C function. The following figure shows an Assembler language initialization function entry point.

```
     PAGE     ,132
     TITLE    DLLSTUB
     NAME     DLLSTUB

     .386
     .387
```

```
     EXTERN   _DLL_InitTerm:NEAR

     END      _DLL_InitTerm
```

The following figure shows a sample initialization/termination function written in C. This code was written using the IBM C Set/2 compiler. If you use another compiler, some of the #pragmas or keywords might need to be changed.

```
/*+-------------------------------------------------------------------+*/
/*| Sample Program 03 : INITTERM.C                                    |*/
/*|                                                                   |*/
/*| COPYRIGHT:                                                        |*/
/*| ----------                                                        |*/
/*| Copyright (C) International Business Machines Corp., 1995         |*/
/*|                                                                   |*/
/*+-------------------------------------------------------------------+*/

#include <stdlib.h>
#include <stdio.h>

/*+-------------------------------------------------------------------+*/
/*| _CRT_init is the C run-time environment initialization function.  |*/
/*|It will return 0 to indicate success and -1 to indicate failure.   |*/
/*+-------------------------------------------------------------------+*/

int _CRT_init (void);

/*+-------------------------------------------------------------------+*/
/*| _CRT_term is the C run-time environment termination function.     |*/
/*+-------------------------------------------------------------------+*/

void _CRT_term (unsigned long);

size_t nSize;
int *pArray;

/*+-------------------------------------------------------------------+*/
/*| _DLL_InitTerm is the function that gets called by the operating   |*/
/*| system loader when it loads and frees this DLL for each process   |*/
/*| that accesses this DLL.  However, it only gets called the first   |*/
/*| time the DLL is loaded and the last time it is freed for a        |*/
/*| particular process.  The system linkage convention must be used   |*/
/*| because OS/2 loader is calling this function.     |*/
/*+-------------------------------------------------------------------+*/

#pragma linkage (_DLL_InitTerm, system)

unsigned long _DLL_InitTerm (unsigned long modhandle, unsigned long flag)
   {
   size_t i;

   /* If flag is zero then the DLL is being loaded so initialization  */
   /* should be performed.  If flag is 1 then the DLL is being freed  */
   /* so termination should be performed.                             */

   switch (flag)
      {
      case 0:
         /* The C run-time environment initialization function must   */
         /* be called before any calls to C run-time functions that   */
         /* are not inlined.                                          */

         if (_CRT_init () == -1)
            return 0UL;

         srand (17);

         nSize = (rand() % 128) + 32;

         printf ("The array size for this process is %u\n",nSize);

         if ((pArray=malloc (nSize * sizeof(int)))==NULL)
            {
            printf("Could not allocate space for unsorted array.\n");
            return 0UL;
            }

         for (i=0; i<nSize; ++i)
            pArray[i] = rand();
```

```
         break;

      case 1:
         printf("The array will now be freed.\n");

         free(pArray);

         _CRT_term(0UL);

         break;

      default:
         printf("flag = %lu\n",flag);
         return 0UL;

      }

   /* A nonzero value must be returned to indicate success. */
   return 1UL;

   }
```

You also can write the initialization/termination function entirely in assembly language, without jumping to a C function. For this case, the library initialization registers are defined as follows:

EIP	Library entry address
ESP	User program stack
CS	Code selector for base of linear address space
DS=ES=SS	Data selector for base of linear address space **Note:** All 32-bit protected memory library modules will be given a GDT selector in the DS and ES registers (ProtDS) that addresses the full linear address space available to an application. This selector should be saved by the initialization routine. Non-protected memory library modules will receive a selector (FlatDS) that addresses the same amount of linear address space as an application's EXE file.
FS	Data selector of the base of the Thread Information Block (TIB)
GS	Is equal to 0
EAX=EBX	Is equal to 0
ECX=EDX	Is equal to 0
ESI=EDI	Is equal to 0
EBP	Is equal to 0
[ESP+0]	Return address to system, and EAX = return code
[ESP+4]	Module handle for the library module
[ESP+8]	Is equal to 0 (for initialization)

A 32-bit library can specify that its entry point address is the 16-bit object code. In this case, the entry registers are the same as for entry to a library using the segmented EXE format. This means that a 16-bit library can be relinked to take advantage of the benefits of the linear EXE format (such as more efficient paging).

The library termination registers are defined as follows:

EIP	Library entry address
ESP	User program stack
CS	Code selector for the base of the linear address space
DS=ES=SS	Data selector for the base of the linear address space
FS	Data selector of the base of the Thread Information Block (TIB)
GS	Is equal to 0
EAX=EBX	Is equal to 0
ECX=EDX	Is equal to 0

ESI=EDI	Is equal to 0
EBP	Is equal to 0
[ESP+0]	Return address to the system
[ESP+4]	Module handle for the library module
[ESP+8]	Is equal to 1 (for termination)

Note: Library termination is not permitted for libraries with 16-bit entries.

Linking at Runtime

So far, the examples in this chapter have used load-time dynamic linking. With load-time linking, OS/2 loads the DLL containing the imported functions when it loads the EXE file. If it cannot find the necessary DLL, it terminates the application and reports the error.

Run-time dynamic linking permits an application to load a DLL into memory when it is required, and to remove the DLL when it is no longer needed. The application uses the DosLoadModule function to load the DLL into memory (if it is not loaded already). If the system cannot find the DLL, the application receives an error value and can take appropriate action. For example, the application might use another DLL or search another directory.

Once the application has loaded the DLL, it can use the DosQueryProcAddr function to obtain a pointer to the required function (or functions). The application then can use the function. When the DLL is no longer required, the application can use the DosFreeModule function to remove the DLL from memory. If there are other applications using the DLL, it remains in memory until the last application frees the DLL.

An application can specify a full path for the run-time DLL. If you specify the full path name, you can have two DLLs with the same name loaded at the same time, as in C:\OS2\DLLFILE.DLL and C:\OS2\DLL\DLLFILE.DLL. If the path is not specified, OS/2 assumes the DLL has the extension .DLL and looks for the file in the directories specified by the LIBPATH environment variable.

The following figure uses the run-time dynamic-linking functions to access the myPuts function in the MYPUTS.DLL dynamic link library.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

VOID (* EXPENTRY myPuts) (PSZ);

VOID main(VOID)
{
    HMODULE hmod;
    ULONG ulErr;
    UCHAR szFailName[CCHMAXPATH];

    ulErr = DosLoadModule(szFailName,          /* failed module name */
        sizeof(szFailName),                    /* size of buffer     */
        "myputs",                              /* name of DLL        */
        &hmod);                                /* module handle here */

    if (ulErr)
        DosExit(EXIT_PROCESS, 1);

    ulErr = DosQueryProcAddr(hmod,    /* DLL module handle           */
        0,                            /* function ordinal value      */
        "myPuts",                     /* function name               */
        (PFN *) &myPuts);             /* address of function pointer */

    if (!ulErr) {

        /* We can use the function now. */

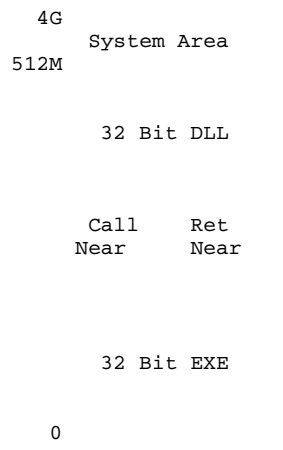
        myPuts("does it work?");

        DosFreeModule(hmod);                /* frees the DLL module */
    }
}
```

Protected Memory Use

OS/2 provides shared library support in the form of 32-bit DLLs. All 32-bit dynamic links or APIs are called using near CALL or RET instructions, so the cost of making dynamic-link calls should be significantly less than the cost of making the comparable calls in the 16-bit version of OS/2, where a far CALL is required. The DLLs execute in the context of the caller.

All 32-bit DLLs are mapped into the appropriate shared memory region of the requesting processes at load time and execute at ring 3 without IOPL. This model's protection characteristics correspond closest to the ring 3 dynamic-linking model in the 16-bit version of OS/2. The following figure shows how 32-bit DLLs are implemented in the linear memory model of OS/2.



A 32-Bit DLL

However, since 32-bit EXE programs can address the entire address space with a 32-bit offset, it is easier for a 32-bit application programmer to potentially cast a bad pointer to data in the shared region than in the 16-bit segmented addressing scheme. Since many subsystems have semaphores and other shared data structures in the shared region, the potential for an inadvertently errant application to affect another process sharing a subsystem becomes an issue in the flat environment. Therefore, OS/2 provides a mechanism for DLLs to protect their critical shared global data regions from 32-bit EXEs. This mechanism prevents a thread in one process from potentially affecting other processes using the same resources (subsystems), or potentially taking down the entire workstation if the compromised subsystem is critical (such as PM).

OS/2 enables existing 16-bit DLLs and new 32-bit DLLs to get their shared global data allocated into a single protected region that is not accessible by 32-bit EXEs, thereby achieving a level of protection. There is no provision for protecting DLLs from each other or from threads executing 16-bit EXE modules. The MEMMAN CONFIG.SYS line supports a "PROTECT/NOPROTECT" option, as follows, for enabling or disabling memory protection:

```
MEMMAN=SWAP,PROTECT
```

If neither PROTECT nor NOPROTECT is specified, the default is protection enabled (PROTECT).

When protection is enabled, the memory manager reserves a 64M region of the linear address space below the 512MB line; this is called the *protected region* . Protected objects are allocated within the protected region. The following types of memory are considered protected:

DLL Global Data	Global data that is part of the DLL image when loaded. This is only global shared data, not instance data. Although DLL code is shared, it is not allocated in the protected region since it is read-only.
DLL Run-Time Shared Data	Global data that is allocated at runtime by a thread executing in DLL code that is a protected API. This includes 16- and 32-bit, named and unnamed, shared memory, and shared memory allocated with DosAllocSeg with the share flag set.

The DS value that is used for the user address space (FlatDS) no longer references a descriptor with a 512MB limit. Instead the system exports another DS value for the user address space called the ProtDS that does have the 512MB limit-the FlatDS limit is reduced by the size of the protected region. When a 32-bit EXE is executing, it runs with the FlatDS and is unable to access protected objects created by 16-bit, 32-bit, or 16- and 32-bit DLLs. If the thread calls a 16-bit DLL API entry point, the DLL will have addressability to the protected region through the LDT. If the thread calls a 32-bit DLL entry point that is protected, the 32-bit DLL entry point contains code to switch to the ProtDS so that the protected region is accessible-the 32-bit DLL switches back to the FlatDS before completing service. A switch on the C compiler is used to generate the code sequence as shown in the following figure.

```
DLLAPI  proc
    push    ds
    push    es
    mov     dx, seg FLAT:DGROUP
```

```
    mov     ds, dx
    mov     es, dx

    .....

    pop     es
    pop     ds
    ret
DLLAPI  endp
```

Although SS is not loaded with the ProtDS, a subsystem that switches stacks to a protected stack must write some assembler code to change ESP-thus the subsystem also should set up SS to be the ProtDS when performing the stack switch.

When protection is not enabled, FlatDS=ProtDS and the code still works the same.

Note: The system currently is not sensitive to whether parameters are being validated relative to the FlatDS or the ProtDS when ring 0 kernel APIs are called. Also the 32 16 thunks do not probe 32-bit parameters before converting them and passing them to a 16-bit DLL.

The grouping of protected allocations can be enabled or disabled on a per DLL basis. For 32-bit DLLs, the Linker uses the PROTECT parameter in the DEF file to provide protection information in the DLL's module flags to the loader. All 16-bit modules requiring protection must be specified with the new PROTECT16 CONFIG.SYS parameter.

```
PROTECT16=DLLNAME1,DLLNAME2,...,DLLNAMEX
```

Notice that the DLL suffix is not required. Only DLL files can get the protection.

DLL Side Effects

Dynamic link routines are not processes. They run on the thread of the calling process and therefore do not own resources. Any resource that they obtain or use is owned by the calling process. Authors of DLLs should be careful not to needlessly allocate resources until the resource is required by the calling process to perform the requested function. They also should free the resource as soon as that resource is no longer required.

A dynamic link routine that obtains and uses resources should attempt to minimize the use of a process's resources. For example, stack space should be conserved. If an application redirects file handle 5 and calls a DLL entry that expects file handle 5 to be an open handle to an associated device driver, unpredictable results can occur.

If the routine opens an abundance of file handles, it might consider increasing the maximum number of file handles, so that the process maximum is not exceeded. However, increasing the maximum number of file handles for a process also increases the maximum number of file handles for all processes created by the current process. This will cause additional memory to be consumed and could cause problems for an application that assumes a limit of 20 file handles. Also, it should be noted that applications have the ability to redirect file handles.

Dynamic link routines also should not make system calls that affect the calling process environment. If a DLL changes a process's current directory, another thread running under the same process could fail a file I/O call if it assumes a given working directory.

Applications and DLLs should not make calls to other DLLs, including system DLLs, within a critical section. Since DLLs can use semaphores to synchronize threads within a process or between processes, calling a DLL within a critical section could cause application deadlocks. This would occur if the DLL requests a semaphore on behalf of the calling thread and another thread within the process owns the semaphore. Because the calling thread is in a critical section and is the only thread within the process that is permitted to execute, the semaphore will never be freed, causing a deadlock.

Summary

There are two types of linking: *static* and *dynamic*. Static linking enables a program's code and data to be contained in a single executable file, enabling the system to load it all into memory at once. Dynamic linking permits several applications to use a single copy of an executable module, since the executable module is completely separate from the applications that use it.

The advantages of dynamic linking are:

- Reduced memory requirements
- Simplified application modification
- Flexible software support

- Transparent migration of functions
- Multiple programming language support
- Application controlled memory usage.

OS/2 provides two types of dynamic linking: *load-time* and *run-time*. In load-time dynamic linking, an application is linked with a library file that contains a record that describes where the routine can be found instead of a file that contains the code for the routine. The DLL can be loaded as soon as a process accesses the DLL or when needed. In run-time dynamic linking, the EXE for an application does not contain a record describing where the external routines can be found. Instead, the application explicitly tells OS/2 when to load and free the dynamic link module.

DLL data can be shared or not shared by all processes that use it.

Error Management

Error checking and error handling is extremely important in a multitasking operating system. The conditions in which an application is executing can change at any time due to the activity of other programs executing concurrently with the application. This section describes the functions that an application can use to manage errors that occur during processing.

The following topic is related to the information in this scetion:

- Exception management

About Error Management

Successful completion of most Control Program functions is indicated by an error return code of 0. In the event of an error, Control Program functions usually return an error code that has a non-zero integer value. The non-zero value equates to a symbolic error identifier in the include file, BSEERR.H. The symbolic identifiers indicate the cause of the error. For example, a return code of 2 from DosOpen equates to the symbolic identifier ERROR_FILE_NOT_FOUND; the cause of the error is that the file being opened cannot be found.

DosErrClass and DosError are supplied to assist in error processing.

- DosErrClass takes as input a non-zero return value that was received from any control-program function. (Any return value other than 0 indicates that an error occurred.) The output is a classification of the error and a recommended action. Depending on the application, the recommended action could be followed, or a specific recovery action could be performed.

- DosError enables an application to prevent OS/2 from displaying a default error message in a pop-up window when either a hard error or a software exception occurs.

Classifying Return Values

When a control-program function has been successfully completed, a return value of 0 is returned to the calling thread. A non-zero return value indicates that an error has occurred.

Each non-zero value corresponds to a symbolic error identifier that indicates the cause of the error. For example, a return value of 2 from DosOpen (indicating that the file was not found) corresponds to the symbolic identifier ERROR_FILE_NOT_FOUND.

DosErrClass helps applications deal with non-zero return values by taking a return value as input and returning both an error classification and a recommended action. Depending on the application, the recommended action could be followed, or a more specific recovery routine could be executed.

Disabling Error Notification

A hard error is typically an error (such as the opening of a disk-drive door while a diskette is being read, or any similar kind of device error)

that cannot be resolved by software. When a hard error occurs, the system default action is to prompt for user input by displaying a message in a pop-up window.

DosError disables the default action, foregoing the displayed message and causing an appropriate return value to be returned to whichever control-program function was running when the hard error occurred. The application must determine the appropriate response by referring to the return value.

DosError also enables the application to disable end-user notification if either a program exception or an untrapped numeric-processor exception occurs. However, if one of these exceptions occurs while user notification is disabled, the application will still be ended.

As with hard errors, the system default is that user notification for these exceptions is enabled.

Using Error Management

OS/2 supplies DosErrClass and DosError for error processing. DosErrClass aids in determining the appropriate action that an application should take in response to an error. DosError enables applications to disable the pop-up windows used by OS/2 to inform the user of a hard-error or an exception.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Classifying Errors

DosErrClass receives a non-zero return value from another control-program function as input. It then classifies the return value, tells where in the system the error occurred, and recommends a corrective action.

In the following example, an attempt is made to delete a nonexistent file. The return value is then passed to DosErrClass so that more information about the error can be obtained, including any corrective actions that can be taken.

```
#define INCL_DOSMISC
#include <os2.h>

#define FILE_DELETE "JUNK.FIL"

ULONG   ulError;
ULONG   ulClass;
ULONG   ulAction;
ULONG   ulLocus;
APIRET  ulrc;

ulError = DosDelete(FILE_DELETE);          /* File name path               */

ulrc = DosErrClass(ulError,                /* Return value to be analyzed  */
                   &ulClass,               /* Error classification         */
                   &ulAction,              /* Recommended corrective action */
                   &ulLocus);              /* Where the error occurred     */
```

When called by a family-mode application, this function can return a valid error classification only for errors that have actually occurred. Also, the classifications of a given return value might not be the same for family-mode and OS/2-mode applications.

Disabling Hard-Error and Exception Messages

DosError disables or enables end-user notification of hard errors, program exceptions, or untrapped, numeric-processor exceptions.

In the following example, pop-up windows for hard errors and exceptions are disabled, then enabled again.

```
#define INCL_DOSMISC   /* Error and exception values */
#include <os2.h>


/*********************************************/
/* use pre-defined constants                 */
/* FERR_DISABLEHARDERR   (0x00000000)        */
/* FERR_ENABLEHARDERR    (0x00000001)        */
/* FERR_ENABLEEXCEPTION  (0x00000000)        */
/* FERR_DISABLEEXCEPTION (0x00000002)        */
/* to create constants needed for DosError calls */
/*********************************************/

#define ENABLE_ERRORPOPUPS  FERR_ENABLEEXCEPTION | FERR_ENABLEHARDERR
#define DISABLE_ERRORPOPUPS FERR_DISABLEEXCEPTION | FERR_DISABLEHARDERR

APIRET ulrc;    /* Return code */

ulrc = DosError(DISABLE_ERRORPOPUPS);   /* Action flag for disable */
ulrc = DosError(ENABLE_ERRORPOPUPS);    /* Action flag for enable  */
```

The action to take is encoded as a binary flag. The following table shows the bit-values and their meanings.

Bit Values to Enable and Disable Hard-Error and Exception Pop-up Messages

Bit	Value	Meaning
0	1	Enables hard-error pop-up messages.
0	0	Disables hard-error pop-up messages.
1	0	Enables exception pop-up messages.
1	1	Disables exception pop-up messages.

If DosError is not called, user notification for hard errors and exceptions is enabled by default.

Exception Management

An *exception* is an abnormal condition that can occur during program execution. Common causes of exceptions include:

- I/O errors
- Protection violations
- Math errors
- Intervention by the user or by another process

Activities that can cause exceptions include:

- Trying to use memory that you do not have permission to access
- Dividing by 0
- The user pressing Ctrl+Break

Exceptions include both unexpected errors (such as a memory protection violation) and expected errors (such as guard-page exceptions). Exceptions can be a *synchronous exception* , that is, caused by an action of the executing thread, or an *asynchronous exception* , caused by an event external to the executing thread (such as the user pressing Ctrl+Break). When an exception is caused by the user pressing Ctrl+Break or Ctrl+C, or by another process issuing DosKillProcess for your process, the exception is called a *signal exception* .

In most cases, the default action taken by OS/2 when an exception occurs is to terminate the application that caused the exception. Rather than having OS/2 default action occur, an application can register its own subroutine to handle exceptions. These routines are called *exception handlers* . Exception handlers enable an application to handle some errors itself, allowing the application to avoid termination (or at least to terminate gracefully).

When exception handlers are registered, they are added to an *exception handler chain* . The chain starts empty and each new handler is added to the head of the chain. Exceptions are passed to the exception handlers in the chain in Last-In-First-Out order, so the last exception handler to be registered is the first one to get an opportunity to handle each exception.

Exception handlers have the capability to complete critical code sections without being interrupted by other asynchronous exceptions; these critical code sections are called *must-complete* sections.

Exception handlers can be removed from the exception handler chains with DosUnsetExceptionHandler. Another way that exception handlers

can be removed from the chain is with an *unwind* operation. When unwinding an exception handler, the exception handler is first called, then removed from the exception handler chain.

The following topics are related to the information in this chapter:

- Memory
- Program execution and control

About Exception Management

A multitasking operating system must manage applications carefully. A serious error (such as an attempt to access protected memory) occurring in one application cannot be permitted to damage any other application in the system. To manage errors that might damage other applications, OS/2 defines a class of error conditions called exceptions and defines default actions for those errors.

When an exception occurs, the default action taken by OS/2 is usually to terminate the application causing the exception (unless the application has registered its own exception handling routines). In some cases, when the exception can safely be ignored, execution is allowed to continue.

Rather than having OS/2 default action occur, an application can register its own *exception handlers* routines. An exception handler routine could be written to correct certain error conditions-when these error conditions occur, the thread's exception handler gets the exception, corrects the condition, and the thread continues executing rather than being terminated immediately by OS/2. OS/2's default action is taken if there are no user-defined exception handling routines or if all user-defined routines return without handling the exception.

An application can use DosSetExceptionHandler to register an exception handling routine. DosSetExceptionHandler takes a pointer to an EXCEPTIONREGISTRATIONRECORD data structure as its only argument. The first field in this data structure is a pointer to the previous exception handler in the chain. This field is maintained by OS/2 and must never be modified by an application. The second field is a pointer to the exception handling routine that will be registered by OS/2.

A single exception handler can be used to handle all the exceptions that you choose to handle. It is not necessary to have a separate exception handler for each exception.

Once an exception handling routine is registered, the system will notify it when an exception occurs. OS/2 sends synchronous exceptions only to the thread causing the exception. An application must register an exception handler for each thread that is handling exceptions. When OS/2 terminates an application, however, a process-termination exception is sent to all threads used by the application to be terminated. When the user presses Ctrl+Break, an asynchronous signal exception is sent only to Thread 1, the main thread, of the executing process.

The exception handling routine is passed the following four parameters that provide exception-specific information:

EXCEPTIONREPORTRECORD
: Describes the exception and its parameters. The first field of this data structure contains the number of the exception that occurred.

EXCEPTIONREGISTRATIONRECORD
: The EXCEPTIONREGISTRATIONRECORD data structure used to initially register the exception handler. This is a microprocessor-specific value.

ContextRecord
: Describes the machine state at the time the exception occurred.

DispatcherContext
: Contains state information on nested exception and collided unwinds. This information must not be modified by the application.

Details of the parameters and data structures can be found in Exception Handler Interface.

OS/2 places the exception handlers for each thread in an *exception handler chain* . Registering an exception handler adds it to the head of the chain.

When an application registers an exception handler, the exception handler is added to the head of the chain. If the application calls a routine in a dynamic link library (DLL), the DLL might register an exception handler in case there is an exception while its code is executing; the DLL deregisters the exception handler before returning control to the application. The DLL's exception handler would be ahead of the application's exception handler in the chain.

Exception handlers in the chain are notified of an exception in Last-In-First-Out (LIFO) order. Thus, if an exception occurs while your thread is executing, the exception handler for your thread is notified of the exception first. If your exception handler chooses to handle the exception, the earlier exception handlers in the chain never see the exception. If your exception handler chooses not to handle the exception, it is passed along to the next earlier exception handler in the chain. If no exception handler in the chain chooses to handle the exception, OS/2 takes the default action for the exception.

If an exception happens while DLL code is executing, and if the DLL's exception handler chooses to handle the exception, your application's exception handlers will never be aware it.

System Exceptions

OS/2 defines a class of error conditions called *system exceptions* , and specifies the default actions that are taken when these system exceptions occur. The default action taken by OS/2 in most cases is to terminate the thread that caused the system exception.

System exceptions include both synchronous and asynchronous exceptions. *Synchronous exceptions* are caused by events that are internal to the execution of a thread. For example, synchronous exceptions could be caused by invalid parameters, or by the request of a thread to end its own execution.

Asynchronous exceptions are caused by events that are external to the execution of a thread. For example, an asynchronous exception can be caused by a user entering a Ctrl+C or Ctrl+Break key sequence, or by a process calling DosKillProcess to end the execution of another process.

The Ctrl+Break, Ctrl+C, and DosKillProcess-generated exceptions are also known as *signals* , or as *signal exceptions* .

OS/2 delivers exceptions that occur in 16-bit as well as 32-bit code. The sequence or hierarchy for delivering exceptions is as follows:

- When an exception occurs in 32-bit code, the system gives control only to the 32-bit exception handlers registered for the current thread. If the thread has not registered any 32-bit handlers, the system default action occurs.
- When an exception occurs in 16-bit code, the system first gives control to the 32-bit exception handlers registered for the current thread. If the exception is not handled by one of these handlers, control is passed to the 16-bit handler, if one exists for the given exception. If there is no 16-bit handler for the exception, the system default action occurs.

Notification of an exception is usually sent only to the thread that caused the exception. However, if a thread uses DosExit to terminate all the threads in the process, notification of the process-termination exception is sent to every thread in the process. The thread that used DosExit gets a XCPT_PROCESS_TERMINATE exception, all the other threads in the process get a XCPT_ASYNC_PROCESS_TERMINATE exception.

Exit-list processing occurs on a per-process basis after a process-termination exception has been delivered to each thread in the process and each thread has finally ended except Thread 1 (the main thread). Therefore, any thread that handles a process-termination exception must eventually end its own execution voluntarily. Otherwise, the process-termination sequence will not conclude properly.

The following tables briefly list the possible exceptions. For more detailed information about the system exceptions, including default system action, parameters, and related trap numbers, see the *Control Program Programming Reference* .

Non-Fatal, Software-Generated Exceptions

Exception Symbolic Constant	Description
XCPT_GUARD_PAGE_VIOLATION	A guard page has been accessed.
XCPT_UNABLE_TO_GROW_STACK	The system is unable to allocate the memory page directly below the guard page just accessed.

Fatal, Software-Generated Exceptions

Exception Symbolic Constant	Description
XCPT_IN_PAGE_ERROR	An I/O error occurred while reading a memory page into memory.
XCPT_PROCESS_TERMINATE	The thread has terminated itself with DosExit.
XCPT_ASYNC_PROCESS_TERMINATE	Another thread in the process has caused the thread to terminate.
XCPT_NONCONTINUABLE_EXCEPTION	An exception handler has attempted to continue execution in response to a non-continuable exception.

XCPT_INVALID_DISPOSITION	An exception handler has returned an invalid value.

Fatal, Hardware-Generated Exceptions

Exception Symbolic Constant	Description
XCPT_ACCESS_VIOLATION	An access violation or page fault has occurred.
XCPT_INTEGER_DIVIDE_BY_ZERO	An attempt to divide by 0 has occurred in an integer operation.
XCPT_FLOAT_DIVIDE_BY_ZERO	An attempt to divide by 0 has occurred in a floating point operation.
XCPT_FLOAT_INVALID_OPERATION	An invalid floating point operation was attempted.
XCPT_ILLEGAL_INSTRUCTION	An attempt was made to execute an instruction that is not defined on the host machine's architecture.
XCPT_PRIVILEGED_INSTRUCTION	An attempt was made to execute an instruction that is not permitted in the current machine mode or that the application does not have permission to execute.
XCPT_INTEGER_OVERFLOW	An integer operation generated a carry-out of the most significant bit.
XCPT_FLOAT_OVERFLOW	A floating point operation generated a resulting exponent that is greater than the magnitude permitted for the operands.
XCPT_FLOAT_UNDERFLOW	A floating point operation generated a resulting exponent that is less than the magnitude provided for the operands.
XCPT_FLOAT_DENORMAL_OPERAND	An attempt was made to perform an arithmetic operation on a denormal operand.
XCPT_FLOAT_INEXACT_RESULT	The result of an operation is not exactly representable in the target format.
XCPT_FLOAT_STACK_CHECK	An illegal stack operation was attempted by the floating point coprocessor.
XCPT_DATATYPE_MISALIGNMENT	An attempt was made to store a data in an

	address that is not naturally aligned on a hardware architecture that does not provide alignment hardware.
XCPT_BREAKPOINT	A breakpoint instruction was executed.
XCPT_SINGLE_STEP	One instruction has been executed in single-step mode.

Fatal Exceptions

Exception Symbolic Constant	Description
XCPT_INVALID_LOCK_SEQUENCE	An invalid operation was attempted within an interlocked section of code.
XCPT_ARRAY_BOUNDS_EXCEEDED	An array index outside its upper and lower boundary was detected.

Unwind Operation Exceptions

Exception Symbolic Constant	Description
XCPT_UNWIND	An unwind operation is in process.
XCPT_BAD_STACK	An EXCEPTIONREGISTRATIONRECORD data structure was reached that is not properly aligned or that is not within the current stack boundaries.
XCPT_INVALID_UNWIND_TARGET	The address of the target EXCEPTIONREGISTRATIONRECORD is below the current stack pointer or not in the exception handler chain.

Fatal Signal Exceptions

Exception Symbolic Constant	Description
XCPT_SIGNAL	A signal was made to your process (usually to stop). All the signal exceptions (Ctrl+Break, Ctrl+C, and XCPT_SIGNAL_KILLPROC) come under this exception.

Signal Exceptions

Signal exceptions are special events sent to a thread when the user presses certain key sequences or when another thread or process

explicitly initiates the exception. There are three types of signal exceptions:

XCPT_SIGNAL_BREAK
: When the user presses Ctrl+Break

XCPT_SIGNAL_INTR
: When the user presses Ctrl+C

XCPT_SIGNAL_KILLPROC
: When another process uses DosKillProcess to send a XCPT_SIGNAL_KILLPROC exception to your process.

Signal exceptions are sent only to Thread 1 (the main thread) in the process receiving the exception. If an exception handler is registered on Thread 1, it must be prepared to receive signal exceptions. The thread 1 exception handler can always ignore the signal exception by returning XCPT_CONTINUE_SEARCH.

If the thread 1 exception handler is to receive signal exceptions, it must use DosSetSignalExceptionFocus to notify OS/2 that it wants to receive the XCPT_SIGNAL_INTR (Ctrl+C) and XCPT_SIGNAL_BREAK (Ctrl+Break) signals. Otherwise, these exceptions are not passed to the exception handler and the default action-to terminate the process-is taken by OS/2. The thread will get XCPT_SIGNAL_KILLPROC signals whether it uses DosSetSignalExceptionFocus or not.

All three of these signals are delivered by a single exception-XCPT_SIGNAL-and the exception handler for Thread 1 can choose to handle none, some, or all of the signals. The signal being sent can be determined by examining the exception information in EXCEPTIONREPORTRECORD.

The following table provides information about each type of signal.

Signal Exceptions

Signal	Symbolic Constant	Description
Ctrl+Break	XCPT_SIGNAL_BREAK	This exception is sent to Thread 1 in the current keyboard-focus process when a Ctrl+Break key sequence is received from the keyboard. The default action taken by OS/2 for this exception is forced process termination.
Ctrl+C	XCPT_SIGNAL_INTR	This exception is sent to Thread 1 in the current keyboard-focus process when a Ctrl+C key sequence is received from the keyboard. The default action taken by OS/2 for this exception is forced process termination.
Kill Process Signal	XCPT_SIGNAL_KILLPROC	This exception is sent to Thread 1 in the process specified when an application uses DosKillProcess. The XCPT_SIGNAL_KILLPROC signal exception results from an action external to the process. The default action taken by OS/2 for this exception is forced process termination.

Handling Signal Exceptions

To handle signal exceptions, a process must first call DosSetExceptionHandler to register a handler for the exceptions. Next, the process must call DosSetSignalExceptionFocus, with the *Flag* parameter set to ON, in order to receive signal exceptions.

After a process calls DosSetSignalExceptionFocus, it remains the signal focus for its screen group until it calls DosSetSignalExceptionFocus again with the *Flag* parameter set to OFF, or until another process in the screen group makes a call to the same function with *Flag* set to ON.

Each call to DosSetSignalExceptionFocus with *Flag* set to ON increments a counter in the per-task data area of the process. Each call with *Flag* set to OFF decrements the counter. When a signal exception occurs, the system checks to see whether the value of the counter is

greater than 0. If it is, the signal is sent.

DosSetSignalExceptionFocus returns ERROR_NESTED_TOO_DEEP if the value of the counter exceeds 65535. If a thread tries to turn off the signal focus when the value of the counter is 0, ERROR_ALREADY_RESET is returned.

All 32-bit exception handlers that are attached to thread 1 of the process will be given an opportunity to handle the signal. If no 32-bit exception handler returns XCPT_CONTINUE_EXECUTION in response to the signal, and if a 16-bit exception handler is registered, then the 16-bit handler for the signal will be executed. If none exists, then the process will be terminated.

In order to continue receiving signals, the process must either return XCPT_CONTINUE_EXECUTION from a 32-bit exception handler, or it must call the 16-bit DosSetSigHandler function, specifying SIG_ACKNOWLEDGE as the value of the *Action* parameter to acknowledge the signal, or it must call DosAcknowledgeSignalException.

The typematic facility of the keyboard could cause a Ctrl+C or Ctrl+Break signal exception to repeat. For this reason, the system holds these exceptions until an exception handler returns XCPT_CONTINUE_EXECUTION, or calls DosAcknowledgeSignalException. However, only one signal exception is actually held; they are not queued by the system.

See Must-Complete Sections for information about how a process can defer the handling of signal exceptions.

Sending Signal Exceptions

A process can send the XCPT_SIGNAL signal exception to another process by calling DosSendSignalException.

In order for the specified process to receive the exception, it must have an exception handler registered for Thread 1, and it must designate itself as the signal focus for its screen group by calling DosSetSignalExceptionFocus.

Presentation Manager applications cannot request exception focus for Ctrl+C and Ctrl+Break. However, establishing an exception handler for Ctrl+C and Ctrl+Break is supported for Vio-Window and full-screen applications.

Raising Exceptions

Asynchronous exceptions that have been deferred in a must-complete section are dispatched automatically by the system when the thread exits the must-complete section. However, a synchronous exception that has been deferred must be raised by calling DosRaiseException.

DosRaiseException can also be used to simulate either an asynchronous or synchronous exception. For example, a floating point emulator (a program that emulates a numeric coprocessor) can use this function to simulate an NPX exception.

Raising a software exception captures the machine state of the current thread in a ContextRecord data structure. The ExceptionAddress field of EXCEPTIONREPORTRECORD is set to the return address of the caller, as are the corresponding fields of the ContextRecord data structure. The system then calls each exception handler on the list, passing each a pointer to EXCEPTIONREPORTRECORD and the created ContextRecord data structures. In the case of a continuable exception for which XCPT_CONTINUE_EXECUTION is returned, DosRaiseException restores the potentially modified context back into the machine before returning. Note that control cannot return to the caller of DosRaiseException if the instruction pointer in ContextRecord has been modified.

The caller of DosRaiseException can set the EH_NONCONTINUABLE bit in the flags field of the EXCEPTIONREPORTRECORD data structure. By doing so, the caller guarantees that it is never returned to after the call to DosRaiseException. Note that once set, the EH_NONCONTINUABLE bit cannot be modified by any exception handler. The system will enforce this.

Following are some possible scenarios that might occur after a call to DosRaiseException has been made:

- If one of the exception handlers returns from a continuable exception with a status of XCPT_CONTINUE_EXECUTION, DosRaiseException returns NO_ERROR to the caller, and the thread resumes execution.
- If one of the exception handlers returns from a noncontinuable exception with a status of XCPT_CONTINUE_EXECUTION, the process is terminated, because it is illegal to return XCPT_CONTINUE_EXECUTION from a noncontinuable exception.
- If none of the exception handlers in the thread's chain of handlers returns with a status of XCPT_CONTINUE_EXECUTION, then the action taken depends on the exception number:
 - If the exception number indicates a user-assigned exception or an unassigned system exception, the process is terminated.
 - If the exception number is assigned to a system exception, and CS:EIP points to 32-bit code, no 16-bit handlers are called and the system default action is taken. Depending on which system exception has been raised, the default action is either to terminate the process, or to continue execution of the thread with NO_ERROR returned to the caller.

- If the exception number is assigned to a system exception that maps to a 16-bit exception and CS:EIP points to 16-bit code, a 16-bit exception handler is called, if one is registered. Otherwise OS/2 takes the default action.

User-Defined Exceptions

Exceptions can also be defined by the application. These are called *user-defined exceptions* (as opposed to system-defined exceptions, which are those exceptions defined by OS/2). Applications can define an exception in the following fashion:

```
#define XCPT_YOUR_EXCEPTION  0xE004ABCD
```

The application then raises the exception, using DosRaiseException:

```
EXCEPTIONREPORTRECORD  ERepRec;

ERepRec.ExceptionNum                = XCPT_YOUR_EXCEPTION;
ERepRec.fHandlerFlags               = 0;
ERepRec.NestedExceptionReportRecord = NULL;
ERepRec.ExceptionAddress            = NULL;
ERepRec.cParameters                 = 0;

DosRaiseException(&ERepRec);
```

The exception handlers in the exception handler chain that are ahead of the application's exception handler will see the exception, but they will not recognize it, so they will return XCPT_CONTINUE_SEARCH. Only the application's exception handler will recognize the exception.

The application's exception handler must return XCPT_CONTINUE_EXECUTION so that the exception will not continue to be passed down the exception handler chain.

Must-Complete Sections

A thread can defer the handling of *asynchronous* exceptions by creating a *must-complete section* . A must-complete section is a section of code that cannot be safely interrupted; it must be allowed to complete its execution even if an asynchronous exception occurs while within its boundaries. For example, a must-complete section can be used:

- When modifying shared-memory data structures that cannot be modified through an atomic operation
- Across database update operations
- During a remote communications operation.

Creating a must-complete section ensures that the execution of critical instructions will be completed and that resources will be cleaned up before the thread ends. When used in conjunction with a *mutual exclusion (mutex) semaphore* , a must-complete section also ensures that a thread will have exclusive access to a resource.

The boundaries of the must-complete section are defined by DosEnterMustComplete and DosExitMustComplete requests. While a thread is executing instructions in a must-complete section, the system will hold *asynchronous exceptions* , which include signal exceptions and asynchronous process terminations.

The system increments a counter each time DosEnterMustComplete is called, and decrements the counter when DosExitMustComplete is called. Any asynchronous exceptions that have been held are dispatched when the counter reaches 0. A count greater than 1 indicates the degree of nesting of the must-complete section. If DosExitMustComplete is called when the count is already 0, ERROR_ALREADY_RESET is returned.

The handling of *synchronous* system exceptions and user-defined exceptions is not deferred by the system. To defer the handling of these exceptions, a procedure typically registers an exception handler (by calling DosSetExceptionHandler) and initializes a local *Raise Exception* flag to 0 before entering the must-complete section. The flag is set to 1, and the information is stored, if the exception handler receives a synchronous exception that it wants to reraise later.

If the value of the *raise exception* flag is 0 after the thread exits from the must-complete section, then no exceptions occurred, and the thread continues its normal operation.

If the value of the flag is 1 after the must-complete section has been completed, then an exception occurred, and the thread must call DosRaiseException to raise the deferred exception for handling.

Note: A thread must not call a function that is outside the scope of the must-complete section (for example, a DLL routine), because an error

in the called routine could cause the process to end without returning. Keep must-complete sections as short as possible.

Unwinding Exception Handlers

In addition to handling exceptions, exception handlers are used to clean up resources during the execution of a nonlocal GOTO instruction or during thread termination. (A nonlocal GOTO instruction jumps to a label outside the current procedure. The label is a procedure address or an address within a procedure that is on the stack, higher in the call frame chain.)

DosUnwindException calls and removes exception handlers from a thread's chain of registered exception handlers up to, but not including, a specified exception handler. This is known as an *unwind operation* . DosUnwindException can also be used to unwind all exception handlers from the thread's exception handler chain and to terminate the thread.

For example, with the C language setjmp() and longjmp() routines, the setjmp() would save the address of the current exception handler structure, along with any other information that is necessary to perform the longjmp() routine. (The address of the current exception handler structure is obtained from the head of the exception handler chain. A pointer to the head of the chain is located in the Thread Information Block.)

The longjmp() routine would initiate the unwind of procedure call frames by calling DosUnwindException and passing to it the saved address of the EXCEPTIONREGISTRATIONRECORD data structure. If the address of the EXCEPTIONREGISTRATIONRECORD data structure is not found in the chain, then the XCPT_INVALID_UNWIND_TARGET exception is raised, and the chain is not unwound.

The machine state at the time of the call to DosUnwindException is captured in ContextRecord. The EH_UNWINDING flag is set in the exception flags field of the EXCEPTIONREPORTRECORD data structure. The EH_EXIT_UNWIND flag is also set if the EXCEPTIONREGISTRATIONRECORD parameter is set to 0 (if the application does not provide its own EXCEPTIONREPORTRECORD parameter OS/2 will construct one). A backward walk through the procedure call frames is then performed to find the target of the unwind operation.

Note: Even though a ContextRecord is used to capture the state of the machine, unwinding is not considered an exception. It is simply delivered through the exception mechanism.

The first parameter to DosUnwindException is the address of an exception handler's EXCEPTIONREGISTRATIONRECORD. DosUnwindException will unwind exception handlers up to, but not including that exception handler. If a -1 is passed to DosUnwindException for this parameter, DosUnwindException will unwind all the exception handlers on the chain. If a 0 is passed to DosUnwindException for this parameter, DosUnwindException will unwind all the exception handlers on the chain and exit.

There is no return from a call to DosUnwindException, unless the stack is invalid. Control is transferred to the specified instruction pointer address. If DosUnwindException encounters an error during its processing, it raises another exception rather than return control to the caller.

If the target call frame is reached and an exit unwind is not being performed (that is, an EXCEPTIONREGISTRATIONRECORD is not 0), then the computed machine state is restored from ContextRecord and control is transferred to the address specified by the target-IP address parameter. Note that the stack pointer is not restored, making it possible to transfer information on the stack. It is the responsibility of the code at the target address to reset the stack pointer as necessary.

DosUnwindException is called with C language calling conventions, which permits the use of a variable number of arguments. Thus, the caller can pass any amount of information on the stack, to be picked up at the target-IP address.

If an exit unwind is being performed (the EXCEPTIONREGISTRATIONRECORD parameter is 0), then all call frames are unwound until the base of the stack is reached.

If the EXCEPTIONREPORTRECORD parameter is specified, then each exception handler encountered during the unwind operation is called, using the specified record. If this parameter is not specified, then DosUnwindException constructs an EXCEPTIONREPORTRECORD that specifies the exception XCPT_UNWIND.

Colliding Unwinds
During an unwind operation, it is possible for one unwind to collide with a previous unwind. This occurs when the scope of the second unwind overlaps the scope of the first unwind. Following are two situations:

- The target frame of the second unwind is a frame that has already been unwound by the first unwind.
- The target frame of the second unwind is a valid frame that is positioned before or after the target frame of the first unwind.

Either of these situations could occur during the following scenarios:

- An unwind handler calls unwind, or
- An unwind handler hits an exception that has called unwind.

In the first scenario, the second unwind is attempting to unwind to an invalid target. This causes the exception

XCPT_INVALID_UNWIND_TARGET to be raised.

In the second scenario, the first unwind is abandoned, and the second unwind continues to its target. The second scenario is far more likely.

Note: A user program that uses high level language exception mechanisms must never call DosUnwindException, because this could create conflicts with the runtime exception strategy of the high level language. Unwind operations in this case are performed through language-supported facilities such as the C language longjmp() routine.

Nested Exceptions

A *nested exception* is an exception that occurs while another exception is being handled.

OS/2 supports nested exceptions because an unhandled exception that occurs in an exception handler should be handled at a higher level-that is, by an ancestor of the procedure that registered the offending handler.

When a nested exception occurs, the EH_NESTED_CALL flag is set in the exception structure to indicate that a nested function call is being made. The normal convention then is for the handler to return immediately without handling the exception if the EH_NESTED_CALL flag is set. Without this flag, it would be easy to create an infinitely recursive situation.

For example, suppose we have the following scenario:

1. Procedure main calls procedure PA, which establishes exception handler HA.
2. Procedure PA calls procedure PB, which establishes exception handler HB.
3. Procedure PB calls procedure PC, which establishes exception handler HC.
4. Procedure PC calls procedure PD.

Now suppose that procedure PD causes an exception. The system refers to the current thread's chain of exception handlers.

Because procedure PD has no handler, the system calls HC, the handler for procedure PC, with the EH_NESTED_CALL flag clear. If handler HC returns CONTINUE_SEARCH, the system calls the next handler in the chain, handler HB, again with the EH_NESTED_CALL flag clear.

Now suppose that exception handler HB causes an exception while it is processing the original exception. The call frames for the procedures are arranged in the following order on the stack:

1. Procedure main
2. Procedure PA
3. Procedure PB
4. Procedure PC
5. Procedure PD
6. OS/2's exception dispatcher
7. Procedure HB, which is the exception handler procedure
8. OS/2's exception dispatcher

The system will now start traversing the exception handler chain again. Exception handler HB could have registered an exception handler, which would be the first handler in the chain. If it had registered a handler, it would be called with the EH_NESTED_CALL flag clear.

The range of the nested exception is exception handlers HC and HB. The end of this range can be determined by the fact that exception handler HB is the currently active handler.

These exception handlers have already been given a chance to handle the original exception. They are now about to be called again in a nested range. Therefore, when handlers HC and HB are called again, they will be called with the EH_NESTED_CALL flag set. If they do not handle the exception, then exception handler HA will be called with the EH_NESTED_CALL flag clear, because it is outside the nested range.

Process Exit Lists

A process executes any routines registered in its exit list (with DosExitList) after the Process Termination exception has been delivered to each thread in the process and after each thread except Thread 1 has finally been terminated. If a thread handles the process termination exception, it must eventually voluntarily terminate, or the exit-list sequence will not finish running properly. Threads must not use DosCreateThread, DosExecPgm, DosStartSession, or DosExit when they are delivered a process termination exception.

Error Pop-Up Screens

Some error conditions, such as general protection violations, cause OS/2 to display a pop-up screen containing information about the error. An application can use DosError to disable error pop-up screens. Typically, a Presentation Manager application would disable error pop-up screens if it sets up its own routines to handle errors that would ordinarily generate pop-up screens.

DosError is also used to control and disable *hard errors* , which usually have to do with reading from and writing to disks.

Exception Handler Interface

Exception handlers are passed four parameters. The interface for writing a 32-bit exception handler is:

```
ExceptionHandler (ExceptionReportRecord,
                  ExceptionRegistrationRecord,
                  ContextRecord,
                  DispatcherContext);
```

The exception handler returns XCPT_CONTINUE_EXECUTION to indicate that the exception has been handled and is to be dismissed, or XCPT_CONTINUE_SEARCH to indicate that the exception has not been handled and is to be passed to the next exception handler on the chain.

Note that there are no invalid exception numbers; if a handler does not recognize an exception number, it simply returns XCPT_CONTINUE_SEARCH.

In addition to handling exceptions, exception handlers are used in *unwind operations* . An unwind operation simply calls and removes exception handlers from the exception handler chain of the thread. Unwind exceptions are not actually being delivered to the handlers, so the individual return codes are irrelevant, and they do not affect the unwind operation.

A single exception handler can be used to handle all the exceptions that you choose to handle. It is not necessary to have a separate exception handler for each exception.

A handler is not required to return to the system; it can handle the exception, and then continue thread execution directly. For example, when an application executes a longjmp(), the C language compiler adds code that essentially performs an unwind operation to clean up the stack. Execution then resumes at the point where the target setjmp() occurred.

For synchronous exceptions, an exception handler can alter the contents of the interrupted thread's context, except for the fields that cannot normally be altered during thread execution. For asynchronous exceptions (signal and termination) changes made to the context of the thread are ignored.

Some exceptions are *continuable* ; if the thread's exception handler handles the exception, execution can continue. If the exception condition is such that execution cannot be continued safely, the exception is said to be *noncontinuable* . If an exception is noncontinuable the EH_NONCONTINUABLE bit is set in the exception structure, and it is an error to indicate the exception has been handled. Returning XCPT_CONTINUE_EXECUTION causes an XCPT_NONCONTINUABLE_EXCEPTION exception to be raised.

Generally, exception handlers can use any function while they are handling an exception. However, while handling a process-termination exception, an exception handler must not call DosCreateThread, DosExecPgm, or DosStartSession, because unpredictable results can occur. A handler also must not call DosExit while handling a process-termination exception, because this request will cause the exception to be dispatched as a nested exception to the current thread's entire chain of handlers.

Exception Handler Parameters

EXCEPTIONREPORTRECORD (EXCEPTIONREPORTRECORD) - input/output
: A pointer to the exception report record, which describes the exception and its parameters.

EXCEPTIONREGISTRATIONRECORD (EXCEPTIONREGISTRATIONRECORD) - input/output
: This is a microprocessor-specific value. For the 80386 microprocessor, this is a pointer to the exception registration record data structure that was used to register the current exception handler.

ContextRecord (CONTEXTRECORD) - input/output
: A pointer to a context record, which describes the machine state at the time the exception occurred.

DispatcherContext (DISPATCHERCONTEXT) - output
: A pointer to a reserved field that receives state information on nested exceptions and collided unwinds. This field returns information to

either the exception dispatcher (in the case of nested exceptions) or to the unwind routine (in the case of collided unwinds). User code must not modify the DispatcherContext field at any time.

When the system's exception handler is called (it is already registered by the exception dispatcher), the exception handler returns NESTED and fills in the DispatcherContext field with the address of the EXCEPTIONREGISTRATIONRECORD corresponding to the exception handler most recently called by the exception dispatcher. This indicates how far the exception dispatcher progressed through the call chain before the nesting occurred. The EH_NESTED_CALL bit is set in the EXCEPTIONREPORTRECORD flags field for each exception handler that is called between handler of the exception dispatcher and the establisher of the most recently called handler.

In the case of a collided unwind, the exception handler registered by the unwind dispatcher will return COLLIDED_UNWIND and the DispatcherContext field will contain a pointer to the target frame of the current unwind.

Exception Management Data Structures

Applications use three data structures for exception management (the DispatcherContext parameter is for system use).

- EXCEPTIONREPORTRECORD data structure
- ExceptionRegistrationRecord data structure
- ContextRecord data structure.

An overview of each of these data structures is presented below.

ExceptionReportRecord Data Structure

The EXCEPTIONREPORTRECORD data structure describes an exception and any additional parameters associated with the exception. The data structure contains fields for the following information:

- Exception number
- Exception flags, describing exception attributes
- A pointer to a nested exception report record, if any
- The address where the exception occurred
- Information for any additional parameters.

For descriptions of the system exceptions see the *Control Program Programming Reference* .

Following are the flags that are set to indicate exception attributes. Only the EH_NONCONTINUABLE flag can be set (but not cleared) by the user. All other flags are set by the system.

EH_NONCONTINUABLE (0x1)
The exception is not continuable, and any attempt to continue causes the exception XCPT_NONCONTINUABLE_EXCEPTION to be raised.

EH_UNWINDING (0x2)
The EXCEPTIONREPORTRECORD data structure describes an exception for which an unwind is in progress.

EH_EXIT_UNWIND (0x4)
An exit unwind operation implies that call frames are being unwound until the base of the stack is reached. Note that EH_UNWINDING is also set.

EH_STACK_INVALID (0x8)
Following are causes for this flag to be set:

- The user stack exceeds the limits specified by the Thread Information Block. Applications can get the Thread Information Block by calling DosGetInfoBlocks.
- A call frame exceeds the stack limits specified by the Thread Information Block.
- A call frame is not aligned on the stack.

This flag is set only when the EXCEPTIONREPORTRECORD is passed to an associated debugger. It is not possible to build exception information on the user's stack when the stack is invalid.

EH_NESTED_CALL (0x10)
EXCEPTIONREPORTRECORD describes an exception raised while the current exception handler was active. That is, a nested exception is in progress, and the current handler was also called to handle the previous exception.

EXCEPTIONREPORTRECORD data structures can be chained together to provide additional information when nested exceptions are raised.

ExceptionRegisterRecord Data Structure

The application is responsible for the creation and registration of the EXCEPTIONREGISTRATIONRECORD data structure. This is the data structure used by the application when it established the exception handler on the chain.

The only restrictions are that each pointer in the linked list must either point directly to the next pointer in the list or contain END_OF_CHAIN (-1), and the field immediately following the pointer field must be the pointer to the exception handler code. No fields other than these two will be examined by OS/2. The application can keep any state information that it chooses in this data structure, as long as it does not alter either of the fields used by the system.

When a procedure begins, it must create an EXCEPTIONREGISTRATIONRECORD on the stack, fill in the pointer to the exception handler routine, and link the data structure to the front of the exception handler chain by calling DosSetExceptionHandler.

Similarly, when the procedure ends, it must remove EXCEPTIONREGISTRATIONRECORD from the chain by calling DosUnsetExceptionHandler. This maintains the necessary frame-exception handler correspondence.

Note:

For the benefit of assembly language programmers, the Thread Information Block (TIB) is located at FS:[0]. This speeds access to the TIB data structure.

Because the FS is used to point to the TIB, applications that use the FS register must restore the original value when they are finished. Exception handling depends on the FS register pointing to the TIB.

EXCEPTIONREGISTRATIONRECORD data structure must be created on the stack of the application. That is, it must be a data structure that is local to the routine that registers the exception handler. It cannot be stored in the application's data segment. The reason for this is that OS/2 must be able to determine the relative ordering of ExceptionRegistration records by examining their addresses.

ContextRecord Data Structure

The ContextRecord data structure describes the machine state at the time of an exception. This data structure is hardware dependent and is not portable. Therefore, as a rule, software should not use the information contained in this data structure. However, hardware dependent code, such as math libraries, can make use of this information to optimize certain operations.

For a hardware-initiated exception, ContextRecord contains the complete machine state at the time of the exception. For a software-initiated exception, ContextRecord contains the machine state at the time the software raised the exception.

The ContextRecord data structure consists of fields for the following:

- General purpose registers
- Segment registers
- The flags register
- The floating point environment and stack.

Note: With asynchronous exceptions (signal and termination exceptions), the context in ContextRecord is read-only. The exception handler can modify it, but the changes with be ignored.

With synchronous exceptions, changes to ContextRecord will be used when the context is restored.

Exception Handler Return Values

Exception handlers can return one of the following values:

XCPT_CONTINUE_SEARCH (0x00000000)

Indicates that the exception has not been handled. The system responds by passing the exception to the previously installed handler in the thread's chain of exception handlers.

XCPT_CONTINUE_EXECUTION (0xFFFFFFFF)
Indicates that the exception has been handled. OS/2 responds by dismissing the exception, restoring the context of the thread, and continuing the execution of the thread.

Using Exception Management

When an exception occurs, the system default action in most cases is to end the application that caused the exception. Instead of having the system default action occur, an application can register its own exception handling routines. Exception handlers can be written to take corrective action so that a thread can continue running rather than being terminated by the system.

If an exception is handled by the application's exception handler, the exception handler must return XCPT_CONTINUE_EXECUTION. If the application's exception handler does not handle the exception, the exception handler must return XCPT_CONTINUE_SEARCH. If all the exception handlers in the exception handler chain return XCPT_CONTINUE_SEARCH, OS/2 takes the default action, which is usually to terminate the process that caused the exception.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Example Exception Handler

This section of the chapter will present a simple exception handler. Because exception handlers are commonly used to handle memory faults, the example will show the exception handler working with a memory fault.

Memory exceptions can occur when an application attempts to access a guard page, attempts to use memory that has been allocated but not committed (a sparse memory object), or when an application attempts to write to memory that has read-only access. Without an application-registered exception handler, some of these exceptions might cause the application to terminate. If the application registers its own exception handler, it can correct the cause of the memory fault and continue to run.

If the application's exception handler handles the exception, it returns XCPT_CONTINUE_EXECUTION. If the routine does not handle the exception, it returns XCPT_CONTINUE_SEARCH so that the exception will be passed to the next handler in the chain.

The following code fragment shows an exception handling routine set up to deal with memory errors:

```
#define INCL_BASE
#define INCL_DOSEXCEPTIONS
#include <os2.h>

#define HF_STDERR 2    /* Standard error handle */

ULONG _cdecl myHandler(PEXCEPTIONREPORTRECORD pERepRec,
                       PEXCEPTIONREGISTRATIONRECORD pERegRec,
                       PCONTEXTRECORD pCtxRec,
                       PVOID p)
{
    ULONG   ulWritten,
            ulMemSize,
            flMemAttrs;

    APIRET  ulrc;

    /* Access violation at a known location */
    if (pERepRec->ExceptionNum == XCPT_ACCESS_VIOLATION &&
        pERepRec->ExceptionAddress != (PVOID) XCPT_DATA_UNKNOWN) {

        /* Page fault */
        if ((pERepRec->ExceptionInfo[0] == XCPT_READ_ACCESS ||
            pERepRec->ExceptionInfo[0] == XCPT_WRITE_ACCESS) &&
            pERepRec->ExceptionInfo[1] != XCPT_DATA_UNKNOWN) {
```

```
            DosWrite(HF_STDERR,
                     "\r\nPage Fault\r\n",
                     15,
                     &ulWritten);


            /* Now query the memory to find out why we faulted. */
            ulMemSize = 1;

            DosQueryMem((PVOID) pERepRec->pExceptionInfo[1],
                        &ulMemSize,
                        &flMemAttrs);

            /* If the memory is free or committed,            */
            /* we have some other problem.                    */
            /* If it is not free or not committed, commit it. */
            if (!(flMemAttrs & (PAG_FREE | PAG_COMMIT))) {
                DosWrite(HF_STDERR,
                         "\r\nAttempt to access uncommitted memory\r\n",
                         40,
                         &ulWritten);

                ulrc = DosSetMem((PVOID) pERepRec->ExceptionInfo[1],
                                 4096,
                                 PAG_DEFAULT |
                                 PAG_COMMIT);

                if (ulrc) {
                    DosWrite(HF_STDERR, "\r\nError committing memory\r\n",
                             27,
                             &ulWritten);

                    return (XCPT_CONTINUE_SEARCH);
                }
                else
                    return (XCPT_CONTINUE_EXECUTION);
            }
        }
    }
    return (XCPT_CONTINUE_SEARCH);
}
```

Registering an Exception Handler

An application uses DosSetExceptionHandler to register its own exception handling routines. More than one routine can be registered; the last routine registered will be called first.

One or more exception handlers can be registered for each thread in a process. Moreover, exception handlers can be specified not only for system exceptions, but also for user-defined exceptions that are anticipated for a particular thread.

Only Process Termination exceptions are sent to all threads in a process. Other exceptions (synchronous exceptions) are sent only to the exception handler registered for the thread where the exception occurred. The application must register an exception handler for each thread that is handling exceptions.

The following code fragment shows how an application registers an exception handling routine:

```
#define INCL_BASE
#define INCL_DOSEXCEPTIONS
#include <os2.h>

ULONG _System myHandler(PEXCEPTIONREPORTRECORD,
                        PEXCEPTIONREGISTRATIONRECORD,
                        PCONTEXTRECORD,
                        PVOID);

VOID main(VOID)
{
    EXCEPTIONREGISTRATIONRECORD xcpthand = { 0, &myHandler };

    DosError(FERR_DISABLEEXCEPTION | FERR_DISABLEHARDERR);

    DosSetExceptionHandler(&xcpthand);
```

```
        /*
         .
         . Other processing occurs here; myHandler will handle the exceptions.
         .
         */

        DosUnsetExceptionHandler(&xcpthand);
    }
```

If a procedure registers an exception handler, it must deregister the handler by calling DosUnsetExceptionHandler before returning.

Note:

A procedure must not call DosSetExceptionHandler if it performs language-specific exception or unwind handling. This restriction is not enforced, but unpredictable results could occur if it is violated.

DosSetExceptionHandler and DosUnsetExceptionHandler provide the portable means of implementing exception handlers. The non-portable approach is taken by directly manipulating the exception handler chain. High level languages generate code that abides by this restriction. Assembly language programmers must assume responsibility for verifying that handler registration and deregistration occur correctly.

EXCEPTIONREGISTRATIONRECORD must be created on the application's stack. That is, it must be local to the routine that registers the exception handler, rather than a global variable. It cannot be stored in the data segment of the program.

Note that in the code fragment above, the declaration is placed inside the braces (see figure below). Therefore *xcpthand* is local to the *main()* routine and is stored on the program's stack.

```
    EXCEPTIONREGISTRATIONRECORD xcpthand = { 0, &myHandler };
```

System Exceptions

The operating system defines a class of error conditions called *exceptions* , and specifies the default actions that are taken when these exceptions occur. The system default action in most cases is to terminate the thread that caused the exception.

Exception values have the following 32-bit format:

```
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

 Sev C        Facility                        Code
```

Sev Severity code. Possible values are described in the following list:

00	Success
01	Informational
10	Warning
11	Error

C Customer code flag.

Facility Facility code.

Code Facility's status code.

Exceptions that are specific to OS/2 Version 2.X (for example, XCPT_SIGNAL) have a facility code of 1.

System exceptions include both synchronous and asynchronous exceptions. Synchronous exceptions are caused by events that are internal to a thread's execution. For example, synchronous exceptions could be caused by invalid parameters, or by a thread's request to end its own execution.

Asynchronous exceptions are caused by events that are external to a thread's execution. For example, an asynchronous exception can be caused by a user's entering a Ctrl+C or Ctrl+Break key sequence, or by a process' issuing DosKillProcess to end the execution of another process.

The Ctrl+Break and Ctrl+C exceptions are also known as *signals* , or as *signal exceptions* .

The following tables show the symbolic names of system exceptions, their numerical values, and related information fields.

Portable, Non-Fatal, Software-Generated Exceptions

Exception Name	Value
XCPT_GUARD_PAGE_VIOLATION	0x80000001
ExceptionInfo[0] - R/W flag	
ExceptionInfo[1] - FaultAddr	
XCPT_UNABLE_TO_GROW_STACK	0x80010001

Portable, Fatal, Hardware-Generated Exceptions

Exception Name	Value	Related Trap
XCPT_ACCESS_VIOLATION	0xC0000005	0x09, 0x0B,
ExceptionInfo[0] - Flags		0x0C, 0x0D,
XCPT_UNKNOWN_ACCESS 0x0		0x0E
XCPT_READ_ACCESS 0x1		
XCPT_WRITE_ACCESS 0x2		
XCPT_EXECUTE_ACCESS 0x4		
XCPT_SPACE_ACCESS 0x8		
XCPT_LIMIT_ACCESS 0x10		
ExceptionInfo[1] - FaultAddr		
XCPT_INTEGER_DIVIDE_BY_ZERO	0xC000009B	0
XCPT_FLOAT_DIVIDE_BY_ZERO	0xC0000095	0x10
XCPT_FLOAT_INVALID_OPERATION	0xC0000097	0x10
XCPT_ILLEGAL_INSTRUCTION	0xC000001C	0x06
XCPT_PRIVILEGED_INSTRUCTION	0xC000009D	0x0D
XCPT_INTEGER_OVERFLOW	0xC000009C	0x04
XCPT_FLOAT_OVERFLOW	0xC0000098	0x10
XCPT_FLOAT_UNDERFLOW	0xC000009A	0x10
XCPT_FLOAT_DENORMAL_OPERAND	0xC0000094	0x10
XCPT_FLOAT_INEXACT_RESULT	0xC0000096	0x10
XCPT_FLOAT_STACK_CHECK	0xC0000099	0x10
XCPT_DATATYPE_MISALIGNMENT	0xC000009E	0x11
ExceptionInfo[0] - R/W flag		
ExceptionInfo[1] - Alignment		
ExceptionInfo[2] - FaultAddr		
XCPT_BREAKPOINT	0xC000009F	0x03
XCPT_SINGLE_STEP	0xC00000A0	0x01

Portable, Fatal, Software-Generated Exceptions

Exception Name	Value	Related Trap
XCPT_IN_PAGE_ERROR	0xC0000006	0x0E
ExceptionInfo[0] - FaultAddr		
XCPT_PROCESS_TERMINATE	0xC0010001	
XCPT_ASYNC_PROCESS_TERMINATE	0xC0010002	
ExceptionInfo[0] - TID of		

terminating thread

XCPT_NONCONTINUABLE_EXCEPTION	0xC0000024
XCPT_INVALID_DISPOSITION	0xC0000025

Non-Portable, Fatal Exceptions

Exception Name	Value	Related Trap
XCPT_INVALID_LOCK_SEQUENCE	0xC000001D	
XCPT_ARRAY_BOUNDS_EXCEEDED	0xC0000093	0x05

Unwind Operation Exceptions

Exception Name	Value
XCPT_UNWIND	0xC0000026
XCPT_BAD_STACK	0xC0000027
XCPT_INVALID_UNWIND_TARGET	0xC0000028

Fatal Signal Exceptions

Exception Name	Value
XCPT_SIGNAL ExceptionInfo[0] - Signal Number	0xC0010003

XCPT_ACCESS_VIOLATION

Access Violation

An access violation exception is generated when an attempt is made either to load or store data in an inaccessible location, or to execute an inaccessible instruction. This exception corresponds to both the Intel 80386 general protection fault (#13), caused by an invalid access attempt; and the page fault (#14), caused by an attempt to access an uncommitted page or a page with incorrect attributes for the desired operation.

Exception Code:
XCPT_ACCESS_VIOLATION (0xC0000005)

Handler Information:
The ExceptionAddress field in the ExceptionReportRecord points to the instruction that caused the exception. This exception is continuable.

Default Action:
The process is ended.

Additional Parameters (2):

Exception Info[0]	Flags
	XCPT_UNKNOWN_ACCESS

(
0
x

0
0
)

XCPT_READ_ACCESS

(
0
x
0
1
)

XCPT_WRITE_ACCESS

(
0
x
0
2
)

XCPT_EXECUTE_ACCESS

(
0
x
0
4
)

XCPT_SPACE_ACCESS

(
0
x
0
8
)

XCPT_LIMIT_ACCESS

(
0
x
1
0
)

Exception Info[1]	FaultAddr The virtual address (if available) of the data that is not accessible, or XCPT_DATA_UNKNOWN.

XCPT_BREAKPOINT

Breakpoint

A breakpoint exception occurs when a breakpoint instruction is executed. This exception is intended for use by debuggers. This exception is continuable.

Exception Code:
 XCPT_BREAKPOINT (0xC0000006)

Default Action:
 The process is ended.

Additional Parameters:
 None.

XCPT_ARRAY_BOUNDS_EXCEEDED

Bounds Check

The bounds check exception corresponds to the Intel 80386 bounds check fault (#5), caused by a BOUND instruction that fails.

Exception Code:
XCPT_ARRAY_BOUNDS_EXCEEDED (0xC0000093)

Handler Information:
The CS:EIP in the exception context structure points to the instruction that caused the exception. This exception is continuable.

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_DATATYPE_MISALIGNMENT

Data-Type Misalignment

A data-type misalignment exception is generated when an attempt is made to load or store data in an address that is not naturally aligned on a hardware architecture that does not provide alignment hardware. For example, 16-bit entities must be aligned on two-byte boundaries, and 32-bit entities must be aligned on four-byte boundaries. This exception does not occur on the Intel 80386 processor. This exception is continuable.

Exception Code:
XCPT_DATATYPE_MISALIGNMENT (0xC000009E)

Default Action:
The process is ended.

Additional Parameters (3):

Exception Info[0]	Read/Write Flag XCPT_READ_ACCESS, or XCPT_WRITE_ACCESS.
Exception Info[1]	Data-type Mask A data-type mask that specifies how many low-address bits must be zero. For example, the data-type mask for a 16-bit entity is one, a 32-bit entity three, and so on.
Exception Info[2]	Virtual Address The virtual address of the misaligned data.

XCPT_FLOAT_DIVIDE_BY_ZERO

Floating Divide-by-Zero

A floating divide-by-zero exception is generated when an attempt is made to divide a floating-point dividend by a floating-point divisor of zero. This exception is continuable.

Exception Code:
XCPT_FLOAT_DIVIDE_BY_ZERO (0xC0000095)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_FLOAT_OVERFLOW

Floating Overflow

A floating overflow exception is generated when the resulting exponent of a floating-point operation is greater than the magnitude allowed for the respective floating point data type. This exception is continuable.

Exception Code:
XCPT_FLOAT_OVERFLOW (0xC0000098)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_FLOAT_UNDERFLOW

Floating Underflow

A floating underflow exception is generated when the resulting exponent of a floating-point operation is less than the magnitude provided for the respective floating-point data type. This exception is continuable.

Exception Code:
XCPT_FLOAT_UNDERFLOW (0xC000009A)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_FLOAT_INVALID_OPERATION

Invalid Floating-Point Operation

This exception usually indicates a programming error corresponding to the invalid floating-point operations defined in IEEE Standard 754. The Intel 80386 processor raises trap #16. This exception is continuable.

Default Action:
The process is ended.

Exception Code:
XCPT_FLOAT_INVALID_OPERATION (0xC0000097)

Additional Parameters:
None.

XCPT_FLOAT_DENORMAL_OPERAND

Denormalized Operand

A denormalized operand exception occurs when the 80387 NPX processor attempts an arithmetic operation on a denormal operand, and the user has not masked off denormal operations. This exception is continuable.

Exception Code:
 XCPT_FLOAT_DENORMAL_OPERAND (0xC0000094)

Default Action:
 The process is ended.

Additional Parameters:
 None.

XCPT_FLOAT_INEXACT_RESULT

Loss of Precision

A loss of precision exception occurs when the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be exactly represented in binary form. For the Intel 80386 and 80387 processors, this corresponds to one of the class of exceptions for which the 80387 processor signals the 80386 processor to raise trap #16. This exception is continuable.

Default Action:
 The process is ended.

Exception Code:
 XCPT_FLOAT_INEXACT_RESULT (0xC0000096)

Additional Parameters:
 None.

XCPT_FLOAT_STACK_CHECK

Invalid Floating-Point Stack Operation

An invalid floating-point stack check is raised when a floating-point processor attempts an illegal operation on a private stack. The Intel 80387 processor maintains eight internal 10-byte "registers" that are individually addressable and yet behave as a push-down stack under the influence of the FLD (push real) and FST (pop real to destination) instructions. Overflow and underflow are checked with each instruction, and this exception is raised when appropriate. This is one of the class of exceptions for which the Intel 80387 processor signals the Intel 80386 processor to raise trap #16. This exception is continuable.

Exception Code:
 XCPT_FLOAT_STACK_CHECK (0xC0000099)

Default Action:
 The process is ended.

Additional Parameters:
 None.

XCPT_ILLEGAL_INSTRUCTION

Illegal Instruction

An illegal instruction exception is generated when an attempt is made to execute an instruction whose operation is not defined for the host machine architecture. On the Intel 80386 processor, this corresponds to the invalid opcode fault (#6), caused by any invalid instruction. This exception is continuable.

Exception Code:

XCPT_ILLEGAL_INSTRUCTION (0xC000001C)

Default action:
The process is ended.

Additional Parameters:
None.

XCPT_PRIVILEGED_INSTRUCTION

Privileged Instruction

A privileged instruction exception is generated when an attempt is made to execute an instruction whose operation is not allowed in the current machine mode. For example, an attempt is made to execute an instruction in user mode that is only allowed in kernel mode. This exception is continuable.

Exception Code:
XCPT_PRIVILEGED_INSTRUCTION (0xC000009D)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_INVALID_LOCK_SEQUENCE

Invalid Lock Sequence

An invalid lock sequence exception is generated when an attempt is made to execute an operation within an interlocked section of code, and the sequence is invalid for the host machine architecture. This exception is continuable.

Exception Code:
XCPT_INVALID_LOCK_SEQUENCE (0xC000001D)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_INTEGER_DIVIDE_BY_ZERO

Integer Divide-by-Zero

An integer divide-by-zero exception is generated when an attempt is made to divide an integer dividend by an integer divisor of zero. On the Intel 80387 processor, this is a divide by zero fault (#0), caused by a DIV or IDIV by zero operation. This exception is continuable.

Exception Code:
XCPT_INTEGER_DIVIDE_BY_ZERO (0xC000009B)

Default action:
The process is ended.

Additional Parameters:
None.

XCPT_INTEGER_OVERFLOW

Integer Overflow

An integer overflow exception is generated when the result of an integer operation causes a carry-out of the most significant bit of the result, which is not the same as the carry-into of the most significant bit of the result. For example, the addition of two positive integers produces a negative result. On the Intel 80387 processor, this corresponds to overflow trap (#4), caused by executing an INTO instruction with the OF flag set. This exception is continuable.

Exception Code:
XCPT_INTEGER_OVERFLOW (0xC000009C)

Default action:
The process is ended.

Additional Parameters:
None.

XCPT_SINGLE_STEP

Single Step

A single-step exception is generated when a trace trap or other single instruction execution mechanism signals that one instruction has been executed. This exception is intended for use by debuggers. This exception is continuable.

Default Action:
The process is ended

Exception Code:
XCPT_SINGLE_STEP (0xC00000A0)

Additional Parameters:
None.

XCPT_GUARD_PAGE_VIOLATION

Guard Page Violation

A guard page violation exception is generated when an attempt is made to load or store data in a location that is contained within a guard page. Memory management software immediately turns the guard page into a demand zero page and initiates a guard page violation exception.

Exception Code:
XCPT_GUARD_PAGE_VIOLATION (0x800001)

Default Action:
Execution continues. If possible, the memory page immediately below the guard page is allocated and marked as a guard page. The higher guard page is marked to no longer be a guard page, and the instruction is restarted. This allows for dynamic stack growth. If it is not possible to allocate another page below the faulting page, an XCPT_UNABLE_TO_GROW_STACK exception is raised. This exception is continuable.

Additional Parameters (2):

ExceptionInfo[0]	Read/Write Flag XCPT_READ_ACCESS, or XCPT_WRITE_ACCESS.
ExceptionInfo[1]	Virtual Address

The virtual address of the data within a guard page.

XCPT_UNABLE_TO_GROW_STACK

Unable to Grow Stack

The default action for a guard page violation is to attempt to allocate another page of memory immediately below the page on which the fault occurred, thereby implementing dynamic stack growth. If this attempt fails, XCPT_UNABLE_TO_GROW_STACK is generated, indicating that the thread has, at most, one more page of stack space available. This exception is continuable.

Exception Code:
XCPT_UNABLE_TO_GROW_STACK (0x80010001)

Default Action:
Execution continues.

Additional Parameters:
None.

XCPT_BAD_STACK

Bad Stack

This exception is raised when an ExceptionRegistrationRecord is reached that is not properly aligned or is not within the current stack boundaries. It is also raised if an unwind target is specified that does not point to an ExceptionRegistrationRecord. This exception is noncontinuable.

Exception Code:
XCPT_BAD_STACK (0xC0000027)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_INVALID_UNWIND_TARGET

Invalid Unwind Target

This exception is raised when the address of the target ExceptionRegistrationRecord is below the current stack pointer. This exception is noncontinuable.

Exception Code:
XCPT_INVALID_UNWIND_TARGET

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_IN_PAGE_ERROR

Page Read Error

A page read error exception is generated when an attempt is made to read a page into memory and an I/O error is encountered. This exception is continuable.

Exception Code:
XCPT_IN_PAGE_ERROR (0xC0000006)

Default Action:
The process is ended.

Additional Parameters (1):

ExceptionInfo[0]	Virtual Address A virtual address within the page that was being read.

XCPT_INVALID_DISPOSITION

Invalid Disposition

This exception is raised when an exception handler returns anything except XCPT_CONTINUE_EXECUTION or XCPT_CONTINUE_SEARCH. This exception is *not* continuable.

Execution Code:
XCPT_INVALID_DISPOSITION (0xC0000025)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_NONCONTINUABLE_EXCEPTION

Continuing a Noncontinuable Exception

This exception is raised when an exception handler returns XCPT_CONTINUE_EXECUTION in response to a noncontinuable exception. This exception is *not* continuable.

Execution Code:
XCPT_NONCONTINUABLE_EXCEPTION (0xC0000024)

Default Action:
The process is ended.

Additional Parameters:
None.

XCPT_PROCESS_TERMINATE

Process Termination

There are two exceptions a thread may receive when it is about to end:

XCPT_PROCESS_TERMINATE, and
XCPT_ASYNC_PROCESS_TERMINATE.

A thread receives XCPT_PROCESS_TERMINATE after it calls DosExit to end itself or the entire process. This exception is *not* continuable.

Additional Parameters:
None.

XCPT_ASYNC_PROCESS_TERMINATE

Asynchronous Process Termination

There are two exceptions a thread may receive when it is about to end:

XCPT_PROCESS_TERMINATE, and
XCPT_ASYNC_PROCESS_TERMINATE.

A thread receives XCPT_ASYNC_PROCESS_TERMINATE when another thread in the process has caused it to end. For example, another thread has called DosExit to end the process, or has not handled a fatal exception, and so on. This exception is continuable.

Additional Parameters (1):

ExceptionInfo[0]	TID The thread identification of the terminating thread.

XCPT_UNWIND

Unwinding

The system fills in an exception number for an unwind if the user chooses not to do so. Note that an ExceptionReportRecord containing XCPT_UNWIND does not indicate that an exception has occurred, but rather that an unwind is in progress.

Exception Code:
XCPT_UNWIND (0xC0000026)

Default Action:
Does not apply.

Additional Parameters:
None.

XCPT_SIGNAL

Signal Exceptions

An OS/2 Version 2.X application may receive three signals:

XCPT_SIGNAL_INTR (Ctrl+C)
XCPT_SIGNAL_KILLPROC (DosKillProcess)
XCPT_SIGNAL_BREAK (Ctrl+Break).

The signal being sent may be determined by examining the exception information in the ExceptionReportRecord.

Exception Code:
XCPT_SIGNAL (0xC0010003)

Default Action:

The process is ended.

Additional Parameters (1):

ExceptionInfo[0] Signal Number

Number	Signal
1	XCPT_SIGNAL_INTR
3	XCPT_SIGNAL_KILLPROC
4	XCPT_SIGNAL_BREAK

XCPT_SIGNAL is called a "signal exception" and is sent only to thread 1 in the process receiving the exception. This is consistent with 16-bit signals, and provides greater consistency in the environment of the process for handling the various asynchronous exceptions. For example, since a repeated typematic Ctrl+C could possibly cause the thread to recursively process the exception and consume stack space without ever being able to handle the first "signal", the exception dispatcher "holds" each exception of the same type until a handler either returns XCPT_CONTINUE_EXECUTION to the exception dispatcher, or the process calls DosAcknowledgeSignalException for that signal. Only one signal or exception is actually held (they are not queued by the system).

DosAcknowledgeSignalException indicates to the system that the process wants to receive the XCPT_SIGNAL_INTR and XCPT_SIGNAL_BREAK signals. Previously, when a process called DosAcknowledgeSignalException the system noted that the process was aware of the particular signal for which it was registering the handler. When a process called DosAcknowledgeSignalException, it became a candidate for the "signal focus" for its session. At any point in time, the focus for a session is the last process to register a signal handler for that signal. When the user presses Ctrl+C on the keyboard, the system delivers an XCPT_SIGNAL_INTR signal to the current keyboard focus. The user could also press Ctrl+Break to deliver an XCPT_SIGNAL_BREAK signal, but this would only work if input were in raw mode.

Note that all exception handlers (on thread 1) must be prepared to "see" signal exceptions. It is always possible that a previous handler has issued DosSetSignalExceptionFocus, or that a Dos16SetSigHandler has been issued by some 16-bit code in the path. They can always be ignored by returning XCPT_CONTINUE_SEARCH to the exception dispatcher. Note that signals result in a call to the 16-bit signal handler (if installed) if all the 32-bit exception handlers return XCPT_CONTINUE_SEARCH.

DosSetSession performs the function of assigning the signal focus exactly as if the application had called DosAcknowledgeSignalException twice, once for each signal. The process calls DosSetSession when it wants to indicate that it expects to receive XCPT_SIGNAL_INTR or XCPT_SIGNAL_BREAK after it has registered an exception handler to process the signal when it comes. Each call to DosSetSession increments a counter in the PTDA of the process. When the system attempts to send XCPT_SIGNAL_INTR or XCPT_SIGNAL_BREAK to a process, it first checks to see if either this counter is greater than zero, or if the process has registered a 16-bit signal handler for that signal. If either of these is true, the signal will be sent. If the process has registered both 16-bit and 32-bit handlers, the 32-bit handlers are called first. If they do not handle the signal, the 16-bit handlers are called. If the 32-bit handlers are called and do not handle the signal, and there are no 16-bit handlers, the process is terminated.

Extended Attributes

OS/2 file systems maintain a standard set of information on file objects. This standard set of information is referred to as *Level 1 file information* . Level 1 file information includes the name and size of the file object, and the date and time the file object was created, last accessed, and last written to.

Applications can attach additional information to a file object in the form of an *extended attribute* (EA). There can be many EAs associated with a file object and, because of their flexibility, almost any information about the file can be stored in one.

The following topics are related to the information in this chapter:

- File Systems
- File Names
- File Management

About Extended Attributes

Level 1 file information is the basic information describing files that is stored by the file system. Level 1 file information includes the size of the file, and the date and time it was created, last written, and last accessed. A subset of this information is typically displayed by entering the DIR command on the OS/2 command line. Applications can obtain Level 1 file information by calling DosQueryPathInfo and DosQueryFileInfo. Applications can set Level 1 File Information by calling DosSetPathInfo and DosSetFileInfo.

Applications can attach additional information to a file object in the form of an *extended attribute* (EA). Extended attributes can be used to describe the file object to another application, to OS/2, and to the file system that is managing that object.

This information can be used to:

- Store notes on file objects (for example, the name of the file creator)
- Categorize file objects (for example, source, samples, icons, bit maps)
- Describe the format of data contained in the file object (for example, a data record)
- Append additional data to the file object.

An application uses extended attributes to provide a description of a file or directory but the application does not place the description in the file or directory itself. Extended attributes associated with a file object are not part of the file object or its data. They are stored separately from the file they are linked to and the file system manages the storage and maintenance of the EA.

Each extended attribute has two parts, a name and a value. The name is a NULL-terminated string; any convenient name can be chosen. EA names are restricted to the same character set as file names.

The value of the EA can be text, a bit map, binary data, anything at all. OS/2 does not check data that is associated with an EA. The application that creates the extended attributes and the applications that read them must recognize the format and meaning of the data associated with a given EA name.

Applications can examine, add, and replace extended attributes at any time. Any application can read the extended attributes by using the DosQueryFileInfo or DosQueryPathInfo function. Applications can use DosFindFirst and DosFindNext to search for files that have specific extended attributes.

A file can have any number of extended attributes. Each extended attribute can be up to 64KB in size. The sum of all extended attributes for a file must not exceed 64KB.

So that extended attribute data can be understood by other applications, conventions have been established for naming EAs and indicating the type of data they contain.

In addition, a set of Standard Extended Attributes (SEAs) have been defined. SEAs define a common set of information that can be associated with most files (for example, file type and file purpose). Through SEAs, many applications can access the same, useful information associated with files.

Applications are not limited to using SEAs to associate information with files. They can define their own application-specific extended attributes.

Extended attributes associated with a file object are not part of the file object or its data.

Extended attributes are supported by the OS/2 High Performance File System (HPFS) and by the OS/2 FAT file system.

Applications define and associate extended attributes with a file object through file system functions. The file system functions that use and manipulate EAs are:

- DosOpen
- DosFindFirst
- DosQueryFileInfo
- DosQueryPathInfo
- DosSetFileInfo
- DosSetPathInfo

Extended Attribute Data Type Conventions

Extended attributes (EAs) can contain any type of data. So that applications can understand the type of information stored in an EA, the first WORD of EA data must specify one of the following data types:

Extended Attribute Data Types

Data Type	Value	Description
EAT_BINARY	FFFE	Binary (non-text) data; the first WORD following the data type specifies the length of the data.
EAT_ASCII	FFFD	ASCII text; the first WORD following the data type specifies the length of the data.
EAT_BITMAP	FFFB	Bit map data; the first WORD following the data type specifies the length of the data.

```
EAT_METAFILE    FFFA   Metafile data; the first WORD following
                       the data type specifies the length of
                       the data.

EAT_ICON        FFF9   Icon data; the first WORD following the
                       data type specifies the length of the
                       data.

EAT_EA          FFEE   ASCII name of another EA that is
                       associated with the file. The contents
                       of that EA are to be included into the
                       current EA. The first WORD following the
                       data type specifies the length of the
                       data.

EAT_MVMT        FFDF   Multi-Valued, Multi-Typed data-two or
                       more consecutive extended attribute
                       values. Each value has an explicitly
                       specified type.

EAT_MVST        FFDE   Multi-Valued, Single-Typed data-two or
                       more consecutive extended attribute
                       values. All values have the same type.

EAT_ASN1        FFDD   ASN.1 field data; an ISO standard for
                       describing multivalue data streams.
```

Values of hex 8000 and up are reserved. Values between hex 0000 and hex 7FFF can be defined by the user.

Symbolic constants are defined in BSEDOS.H and BSEDOS.INC.

In all cases, the length specifies the number of bytes of data. Other values for data types, in the range hex 0000 through hex 7FFF, can be used for user-defined extended attributes.

All user-defined data types should be length-preceded, meaning that a WORD indicating the length of the data (in bytes) precedes the data.

For example, here is how to represent the string "Hello":

```
    EAT_ASCII    0005    Hello
```

Multi-Value Data Type Fields

In many cases, it is desirable for extended attributes (EAs) to store more than a single piece of information. For example, an extended attribute can store a list of names to which a document was sent. The multi-value formats specify how individual pieces of data are stored.

Data entries are length-preceded, making it easy to traverse a multi-valued list.

In order to allow EAs of different code pages, multi-valued EAs include a field in which the EA's code page is specified. For example, the code page field could be used to indicate that the comments for a Kanji file are written in Spanish. If this value is 0, the file default is assumed. (Code page data is for use by applications. OS/2 does not examine or use EA code page information.)

When the concept of a default applies to a multi-valued EA, the first entry in the list is assumed to be the default. For example, suppose an EA entry contains the strings "Text" and "C Code". "Text" is considered the default type. If "C Code" were the first entry in the list ("C Code" then "Text"), then "C Code" would be considered the default type.

There are three multi-valued EA data types:

- Multi-Valued, Multi-Typed Data
- Multi-Valued, Single-Type Data
- ASN.1 Data

Multi-Valued, Multi-Typed Data Type

This data type indicates that the value of a single extended attribute (EA) contains several pieces of information, each of a different data type. It is formatted as follows:

```
EAT_MVMT  Codepage  NumEntries  [DataType Data] ...
WORD      WORD      WORD         WORD
```

The first word indicates that the EA value is multi-valued, multi-typed. The second word indicates the code page associated with the language in which the EA value is written. The third word indicates the number of entries contained in this EA value. The next word indicates the data type for the first entry in this EA value, followed by the data for the first entry. The next word, if any, indicates the data type for the second entry in this EA value, followed by the data for the second entry. The pattern repeats- data type, followed by data-for any remaining entries.

For example, an extended attribute can have the following value:

```
EAT_MVMT  0000  0002  EAT_ASCII   000A Hello John
                      EAT_BINARY  0003 0x12 0x21 0x34
```

This is a multi-valued extended attribute with two entries, using the default code page. The first entry is the string "Hello John", and the second is the binary data 0x12 0x21 0x34.

Whether or not the data is length-preceded is a function of the data type.

Multi-Valued, Single-Type Data Type

This data type indicates that the value of a single extended attribute (EA) contains several pieces of information, each of the same data type. For example:

```
EAT_MVST  Codepage  NumEntries  Data_Type  [data] ...
WORD      WORD      WORD        WORD
```

The first word indicates that the EA value is multi-valued, single-typed. The second word indicates the code page associated with the language in which the EA value is written. The third word indicates the number of entries contained in this EA value. The next word indicates the data type of all the entries contained in this EA value, followed by the data for all entries.

For example, the following EA value contains three ASCII names:

```
EAT_MVST 0000 0003 EAT_ASCII 0004 Mark
                             0005 Ellen
                             0003 Liz
```

Each name string is preceded by the length of the string. Whether or not the data is length-preceded is a function of the data type.

ASN.1 Data Type

This data type indicates that the extended attribute uses the ASN.1 ISO standard to describe a multi-valued data stream.

Including One Extended Attribute in Another

Extended attributes (EA) can contain pointers to data stored in other places.

This data type indicates that the data contained in another EA associated with the file object should be included into the current EA.

For example, the following EA value contains the string "Hello", followed by the data in the EA named AB.STUFF, followed by the string "Bye".

```
EA_MVMT 0000 0003 EAT_ASCII 0005  Hello
                  EAT_EA    0008  AB.STUFF
                  EAT_ASCII 0003  Bye
```

Extended Attribute Naming Conventions

Because many applications use text, bit maps, and other binary data in extended attributes, standard names have been adopted to help identify these formats. An application is not limited to these Standard Extended Attributes but should use them when many applications will be accessing the same data.

Standard Extended Attributes (SEAs) have a dot (.) as a prefix. This identifies the extended attribute as a SEA. The leading dot is reserved, so applications should not define extended attributes that start with a dot. Also, extended attributes that start with the characters $, @, &, or + are reserved for system use.

To ensure that its extended attributes are unique, an application should use the name of the company and the name of the application (or suitable abbreviations of each) as a prefix for application-specific extended attributes.

For example, Company A has an OS/2 Application, B, that defines extended attributes STUFF, MORE_STUFF, and STILL_MORE_STUFF for its file objects. The names of these extended attributes could be represented by the following entry:

```
AB.STUFF    AB.MORE_STUFF    AB.STILL_MORE_STUFF
```

Standard Extended Attributes

The name of a SEA has a dot (.) as a prefix. This identifies the extended attribute as a SEA.

The values of Standard Extended Attributes can be multi- or single-valued, with formats following the data type conventions discussed previously.

Where entries for Standard EAs consist of ASCII characters, case is important.

The Standard EAs that have been defined are:

.ASSOCTABLE
.CLASSINFO
.CODEPAGE
.COMMENTS
.HISTORY
.ICON
.ICON1
.ICONPOS
.KEYPHRASES
.LONGNAME
.PREVNAME
.SUBJECT
.TYPE
.VERSION

Any extended attribute that is defined by the system starts with a period. This allows our software vendors to freely define new extended attributes without conflicting with ours, as long as they follow the convention that non-system extended attributes don't start with a period.

The .TYPE and .ASSOCTABLE extended attributes (EA) are two of the most useful SEAs.

The .TYPE extended attribute indicates what type of data is in a file. It also implies what programs can edit the file, and what icon is to be used for the file. OS/2 can use the .TYPE EA to determine a default application to run and a default icon for a file of a particular type (if there is a .ICON EA, it will be used instead of the icon associated with a particular data type).

The .ASSOCTABLE extended attribute allows an application to indicate the type, extension, and icon for data files that it recognizes. It also contains an ownership flag. This data can be automatically installed by OS/2. When your program recognizes files created by other programs, you might want to install .ASSOCTABLE EA entries for those other programs.

The .ASSOCTABLE Standard Extended Attribute

The .ASSOCTABLE extended attribute (EA) contains information that associates data files with the applications that create them or that know how to use them. The .ASSOCTABLE extended attribute enables an application to indicate the type, extension, and icon for the data files it recognizes. The .ASSOCTABLE EA also contains an ownership flag. This tells OS/2 which application to run when the user double-clicks the mouse on a given data file.

Because programs can understand and reference data files generated by other programs, this EA can be used to link a program with those files.

The name of this EA consists of the string ".ASSOCTABLE". The value of this EA contains application information and consists of multi-valued, multi-typed fields that link the application with:

- the file type (that is, the value of a .TYPE EA),
- the file extension,
- and icon data for data files that it generates or references. The .ASSOCTABLE EA associates icons by file-type. The data file's file-type is indicated in the .TYPE EA, or, if the data file does not have a .TYPE EA, by the extension.

This data can be installed automatically by OS/2.

The format of the EA is as follows.

```
EAT_MVMT 0000 0004 EAT_ASCII  .TYPE name
                   EAT_ASCII   file extension
                   EAT_BINARY  flags
                   EAT_ICON    icon data
```

The source for the .ASSOCTABLE EA is contained in the resource file for an application. The .ASSOCTABLE EA is created using the Resource Compiler from a table with the following form:

```
ASSOCTABLE  assoctable -id
BEGIN
association_name,[extension],[flags], [icon_filename]
association_name,[extension],[flags], [icon_filename]
 .
 .
 .
END
```

The *association_name* is the name of a file type that the Resource Compiler understands. (This is the same name found in the .TYPE field of data files.)

The *extension* is the three letter file extension that is used to identify files of this type, if they have no .TYPE EA entry. (Three letter extensions should be used so that FAT file systems can make use of this EA). This field can be empty.

The *icon_filename* is the name of the file that contains the icon that is to be used to represent this file type. (This field can also be empty.)

The .ASSOCTABLE *flag* indicates that the program is the default application for data files with the specified type. This determines the program OS/2 will start when the file is double-clicked with the mouse.

If more than one program has marked itself as the EAF_DEFAULTOWNER for a particular data file .TYPE, OS/2 will not know which program to run when the file of this .TYPE is double-clicked on with the mouse. If no program is marked as the EAF_DEFAULTOWNER for a particular data file .TYPE, OS/2 will be similarly confused. In both cases, OS/2 provides the user with a list of applications that understand the file .TYPE, regardless of whether the application is the owner or not. The user selects the program to run from this list.

The *flag* entry indicates whether the application owns the file or merely recognizes the .TYPE. If this flag is set, the entry describing data files of this type cannot be edited. This flag is specified if a previously defined icon in the ASSOCTABLE is to be reused. Entries with this flag set

have no icon data defined. The icon used for this entry will be the icon used for the previous entry.

EAF_ flags can be ORed together when specified in the ASSOCTABLE. The EAF_ flags are defined in PMWIN.H and PMWIN.INC.

.ASSOCTABLE Example
For example, My_Company's application, My_Application, generates or references data files that have the following .TYPE names:

```
My_Company My_Application documentation
My_Company My_Application macros
My_Company My_Application spreadsheet
My_Company My_Application chart
Your_Company Your_Application forecast
```

The source for the .ASSOCTABLE extended attribute in the resource file for My_Application could look like the following.

```
ASSOCTABLE
BEGIN
"My_Company My_Application documentation", "DOC", EAF_DEFAULTOWNER, My_App.ICO
"My_Company My_Application macros", "MAC", EAF_DEFAULTOWNER+EAF_REUSEICON
"My_Company My_Application spreadsheet", "SPR", EAF_DEFAULTOWNER+EAF_REUSEICON
"My_Company My_Application chart", "CHT", EAF_DEFAULTOWNER+EAF_REUSEICON
"Your_Company Your_Application forecast", "FOR", 0
END
```

My_Application can load and use some files generated by Your_Application. However, because My_Application is not the default owner of those files, OS/2 does not run My_Application when the user double-clicks on the files with the mouse.

The following example illustrates how the value of the .ASSOCTABLE EA for My_Application might look. It is a multi-valued, multi-typed EA with five multi-valued, multi-typed entries (one for each file type referenced or generated by the application).

```
EAT_MVMT   0000 0005          ; There are 5 associated file types

EAT_MVMT   0000 0004          ; Description of 1st associated file type
EAT_ASCII  0027 My_Company My_Application documentation  ; File type
EAT_ASCII  0003 DOC           ; File extension
EAT_BINARY flags              ; Flags
EAT_ASCII  icon data          ; Physical icon data

EAT_MVMT   0000 0004          ; Description of 2nd associated file type
EAT_ASCII  0020 My_Company My_Application macros
EAT_ASCII  0003 MAC
EAT_BINARY flags
EAT_ICON   icon data

EAT_MVMT   0000 0004          ; Description of 3rd associated file type
EAT_ASCII  0025 My_Company My_Application spreadsheet
EAT_ASCII  0003 SPR
EAT_BINARY flags
EAT_ICON   icon data

EAT_MVMT   0000 0004          ; Description of 4th associated file type
EAT_ASCII  001F My_Company My_Application chart
EAT_ASCII  0003 CHT
EAT_BINARY flags
EAT_ICON   icon data

EAT_MVMT   0000 0004          ; Description of 5th associated file type
EAT_ASCII  001F Your_Company Your_Application forecast
EAT_ASCII  0003 FOR
EAT_BINARY flags
EAT_ICON   icon data
```

The .CLASSINFO Standard Extended Attribute

The .CLASSINFO extended attribute saves the instance data for a file system object.

The .CODEPAGE Standard Extended Attribute

The .CODEPAGE extended attribute (EA) contains the code page for the file. If this extended attribute is not provided, the code page of the file is the system default or is defined by the application.

The code page of the EA data associated with the file is assumed to be that of the file, unless the EA entry is specifically overridden in the code page field in the multi-valued extended attribute data type.

The .COMMENTS Standard Extended Attribute

The .COMMENTS extended attribute (EA) contains miscellaneous notes or reminders about the file (for example, peculiarities, restrictions, or requirements).

The name of this EA consists of the string ".COMMENT". The value of this EA consists of miscellaneous notes and can be multi-valued and of any type.

The .HISTORY Standard Extended Attribute

The .HISTORY extended attribute (EA) contains the modification history for a file object, indicating the author of the file and all subsequent changes. Each entry is separate field in a multi-value field and consists of be ASCII characters only.

The name of this EA consists of the string ".HISTORY". The value of this EA contains the modification history for a file object and can be multi-valued, with each action entry described in a separate field.

Each entry in the .HISTORY field has the following format:

```
PERSON  ACTION(created, changed or printed)  DATE
```

For example, the following .HISTORY extended attribute contains two entries:

```
EAT_MVMT 0000 0002
         EAT_ASCII 0017 Joe    Created  2/10/88
         EAT_ASCII 0017 Harry  Changed  2/11/88
```

This extended attribute can potentially become quite large. To avoid unwanted growth, an application can let the user decide when an entry should be added to this extended attribute. For example, there are some cases when it is important to note when a document is printed. However, it is probably unnecessary to note it every time the file is printed.

The .ICON Standard Extended Attribute

The .ICON extended attribute (EA) specifies the icon to be used for the file representation, for example when the application is minimized. This extended attribute contains the physical icon data used to represent the file object.

If there is no .ICON EA, OS/2 can use the .TYPE entry to determine a default icon to use for the particular file. If there is an .ICON entry, however, it is used instead of the default icon.

The name of this EA consists of the string ".ICON". The value of this EA contains the physical icon data and has the following format:

```
EAT_ICON  data_length  data
```

```
WORD       DWORD
```

The data is of type BITMAPARRAYFILEHEADER and is used to specify an array of one device-dependent and one device-independent icon bit maps. The GpiLoadBitmap and WinLoadPointer functions support this icon file format.

It is best to provide as much icon information as possible. Ideally, an icon should be 64-by-64 bits in 8-color, device-independent format.

The Icon Editor is used to create the icon, which is saved in an icon file. The .ICON extended attribute for an application is created by the Resource Compiler as part of the compile process by specifying the DEFAULTICON keyword, as in:

```
DEFAULTICON <filename.ico>
```

This keyword uses the icon definition contained in the specified icon file (FILENAME.ICO) to create the .ICON EA for the application.

Applications store the binary icon data in this extended attribute. To install icons for data files, the applications can use the .ASSOCTABLE extended attribute, or DosSetPathInfo.

The .ICON1 Standard Extended Attribute

The .ICON1 extended attribute stores the animation icon (for example, the open folder icon) for a folder.

The .ICONPOS Standard Extended Attribute

The .ICONPOS extended attribute saves the icon positioning information for a folder.

The .KEYPHRASES Standard Extended Attribute

The .KEYPHRASES extended attribute (EA) contains key text phrases for the file. Such phrases can be used in performing a database-type search or in helping the user understand the nature of the file.

The name of this EA consists of the string ".KEYPHRASES". The value of this EA consists of key phrases in ASCII.

Key phrases are represented as ASCII characters. Multiple key phrases can be stored in the value of this extended attribute, each stored in a separate entry in a multi-valued field.

For example, the following extended attribute contains three key phrases:

```
EAT_MVST 0000 0003 EAT_ASCII 0008 ABC Inc.
                   EAT_ASCII 000A Salesman A
                   EAT_ASCII 000F Product X sales
```

If there is more than one key phrase, each should be stored in a separate entry in a multi-value field.

The .LONGNAME Standard Extended Attribute

When an application attempts to write a file with a long name to a file system that does not support long names, it must generate a short name for the file. The application should notify the user of the new short name and save the original (long) name in the .LONGNAME extended attribute.

When a file is copied from a system that uses short names to a system that uses long names, the application should check the .LONGNAME extended attribute. If a value is present, the application should rename the file to the long name, then remove the .LONGNAME extended

attribute.

The .PREVNAME Standard Extended Attribute

The .PREVNAME extended attribute saves the old class name when the user requests that an object, which is a descendent of WPDataFile, becomes another subclass of WPDataFile.

The .SUBJECT Standard Extended Attribute

The .SUBJECT extended attribute (EA) contains a brief summary of the content or purpose of the file object it is associated with.

The name of this EA consists of the string ".SUBJECT". The value of this EA consists of a single-valued ASCII string that contains the purpose of the file object.

The length of this field must be less than 40 characters.

The .TYPE Standard Extended Attribute

The .TYPE extended attribute (EA) indicates the file-type of the file object it is associated with. It is similar to a file name extension.

The name of this EA consists of the string ".TYPE". The value of this EA contains the file object's file-type. The following file types are predefined:

Plain text
OS/2 command file
DOS command file
Executable
Metafile
Bit map
Icon
Binary data
Dynamic link library
C code
Pascal code
BASIC code
COBOL code
FORTRAN code
Assembler code
Library
Resource file
Object code

Data files only require identification of the file type. For data files without EAs, the file type is derived from the file extension, if there is one.

File object types are represented as length-preceded ASCII strings, uniquely identifying the file object's type. This identifier is referenced within the application's .ASSOCTABLE EA in order to bind the data file type to the application. It is important that this name be a unique identifier because all file type names are public data. For example, if application A and application B both had a type name of SPREADSHEET, the filing system would not be able to identify A's SPREADSHEET from B's SPREADSHEET.

The recommended convention for defining file object types is:

- Company_name
- Application_name
- Application-specific_name

For example, spreadsheet files generated by My_Application written by My_Company might have a file object type of the following.

```
My_Company My_Application Spreadsheet
```

Type names must be ASCII characters and case is significant.

Note: The performance of extended attributes is dependent on the file system. Because some file systems store extended attributes in first-in/first-out (FIFO) order, it is important to write the .TYPE entry first so OS/2 can access that information quickly.

The .VERSION Extended Attribute

The .VERSION extended attribute (EA) contains the version number of the file format, as shown below.

```
My_Application 1.0
```

The name of this EA consists of the string ".VERSION". The value of this EA contains the file object version number. This attribute can be ASCII or binary. Only the application that created the file object should modify the value of this EA. It can also be used to indicate an application or dynamic link library version number.

Managing Extended Attributes

An application can create, query, and set extended attributes (EAs) for any file object.

The application can define extended attributes for a file when the file is created with DosOpen. Similarly, the application can define EAs for a directory when the directory is created using DosCreateDir.

An application can define EAs for existing file objects by referencing the file object by its handle and calling DosSetFileInfo, or by referencing the file object by its name and calling DosSetPathInfo.

An application can examine the EAs for a file object by referencing the file object by its handle and calling DosQueryFileInfo, or by referencing the file object by its name and calling DosQueryPathInfo. The application can also call DosEnumAttribute, using either the file object's handle or its name, to get information about the file object's EAs.

In addition, an application can search for file objects and specify that certain EAs be returned by calling DosFindFirst.

Controlling Access to Extended Attributes

Like the file objects they are associated with, extended attributes (EAs) can have more than one process accessing them at the same time. This means that one process could be querying EAs for a file object, while another is setting EAs for the same file object.

In addition, operations on EAs are not atomic. That is, a query or set operation might not complete before another query or set operation is performed on the same object. If an error occurs before an entire list of EAs has been set, all, some, or none of them may actually remain set on the file object. This means that EAs may not remain in a consistent state unless the order in which the operations are performed can be guaranteed.

Sharing protection is provided so that unpredictable results do not occur during multiple simultaneous operations on extended attributes. EA manipulation is associated with the access permission to the related file object (file or directory).

Handle-based access permission is controlled by the sharing/access mode of the associated file object:

- If the file object is open for read access, querying EAs (using DosQueryFileInfo) is permitted.
- If the file object is open for write access, setting EAs (using DosSetFileInfo) is permitted.

Path-based access permission is controlled by adding the file object to the sharing set for the duration of the call: an application requires read access and file-sharing permission must be set to deny-write.

- For setting EAs (using DosSetPathInfo), an application requires write access and file-sharing permission must be deny-read-write.

Note: The functions that set and query EAs fail if another process holds conflicting sharing rights to the file object.

No explicit EA sharing is performed for DosEnumAttribute. Implicit sharing exists if the caller passes the handle of an open file, since sharing access to the associated file is required to modify its EA. No sharing is performed if the caller passes the path name. This means that if some other process is editing the EAs, and changes them between two calls to DosEnumAttribute, inconsistent results might be returned (for example, the same values might be returned twice, some values might be missed, and so on).

To prevent the modification of EAs for the handle case, the file should be opened in deny-write mode before calling DosEnumAttribute. To prevent the modification of EAs for the path name case, the file should be open in deny-write mode before calling DosEnumAttribute. For the directory name case, no sharing is possible.

Extended Attribute Data Structures

There are a series of data structures through which OS/2 functions manipulate extended attributes (EAs) for applications:

- Full EAs (FEA2s)
- A list of full EAs (FEA2LIST)
- Get EAs (GEA2)
- A list of get EAs (GEA2LIST)
- EAOP2s

Full Extended Attribute (FEA2) Data Structure
A *full EA* (FEA2) data structure contains the extended attribute name and value. The name length does not include the trailing NULL. The characters that form the name are legal file name characters.

An *FEA2LIST* is a list of FEA2 structures, preceded by the length of the list (including the length itself). FEA2Lists are used for querying, adding, deleting, or changing EAs. They are required input parameters for the functions that create or set extended attributes (DosSetFileInfo and DosSetPathInfo). They are required output parameters for the functions that query extended attributes (DosQueryFileInfo, DosQueryPathInfo, and DosEnumAttribute).

FEA2 data structures include the lengths of the extended attribute's names and values. EA name lengths of 0 are illegal and cause errors to be returned by EA functions. An EA value length of 0 has special meaning:

- Setting an EA with a value length of 0 in the FEA2 data structure causes that attribute to be deleted, if possible.
- Getting an EA with a value length of 0 in the FEA2 data structure indicates that the attribute is not present.

Get Extended Attribute (GEA2) Data Structure
A Get EA (GEA2) is an extended attribute name. The name length does not include the trailing NULL.

A GEA2LIST is a list of GEA2 structures, preceded by a length of the list (including the length itself). GEA2Lists are used for retrieving the values for a particular set of extended attributes. They are required input parameters for the functions that query extended attributes (DosQueryFileInfo, DosQueryPathInfo, and DosEnumAttribute).

Note: GEA2 data structures include the lengths of extended attribute's names and values. EA name lengths of 0 are illegal and cause errors to be returned by EA functions. An EA value length of 0 has special meaning. Setting an EA with a value length of 0 in the FEA2 data structure causes that attribute to be deleted, if possible. Getting an EA with a value length of 0 in the FEA2 data structure indicates that the attribute is not present.

Extended Attribute Operation (EAOP2) Data Structure
An extended attribute operation (EAOP2) data structure consists of a GEA2LIST, an FEA2LIST, and an error field. All extended attribute manipulation is performed using this structure. Before calling an extended attribute function, an application must define an EAOP2 structure, with the GEA2LIST and FEA2LIST appropriately defined.

The use of GEA2LIST and FEA2LIST for each function is described further in the *Control Program Programming Reference* .

Preserving Extended Attributes

Extended attributes (EAs) are supported in OS/2's FAT file system and High Performance File System (HPFS). Extended attributes are not supported by the FAT file system used in versions of OS/2 prior to Version 1.2, nor are they supported in the DOS operating system.

EAs associated with a file object are not part of a file object or of its data, but are maintained separately and managed by the file system that manages that object. EAs reside on the volume on which the file object resides and are connected to their file object by pointers from the file

object.

Therefore, EAs belonging to a file object are lost when moving that file object from a storage device managed by an OS/2 FAT or installable file system to a storage device managed by the old FAT file system.

EAs can be lost under other circumstances, as well:

- When a program that does not recognize EAs (for example, an editor written for DOS) performs a non-truncating open on the files with which the EAs are associated
- When the files they are associated with are sent over COM links

So that files with EAs can be manipulated under these circumstances, without losing their EAs, OS/2 provides a utility called EAUTIL EAUTIL enables users to separate extended attributes from the specified file object, optionally disassociating them from the file object. The extended attributes are placed into a specified HoldFile. This utility also enables users to reattach the separated extended attributes with their file objects.

EAUTIL can be applied to subdirectories, as well as to files. It does not support global file name characters in parameters; it operates on a single file object at each invocation.

Users can use EAUTIL to:

- Strip EAs off files to be edited with a program that does not recognize EAs.
- Place the EAs into normal files so they can be dealt with by old tools (such as backup, restore, and so on) or easily transmitted.
- Reattach EAs to files after the files have been brought back from the systems where EAs are not supported.

Protecting Extended Attributes

Programs written for releases of OS/2 and DOS that do not support extended attributes (EAs) will tend to lose EAs simply because they do not know that they exist. OS/2 provides some controls that prevent old programs from destroying critical data without unduly restricting their activities. This is done by classifying programs and marking the extended attributes that are associated with files.

Programs are classified as:

- Those that recognize EAs. These include OS/2 Version 1.2 and later programs.
- Those that do not recognize EAs. These include programs written for releases of OS/2 and DOS that do not support EAs.

EAs associated with files are marked as critical or non-critical. Programs that do not recognize EAs are not permitted to manipulate files that have critical EAs associated with them. This protection does not apply to directories. EAs associated with directories cannot be marked as critical.

Critical Extended Attributes
Extended attributes (EAs) are non-critical by default. A non-critical EA is one that is not necessary to the functionality of the application. That is, if a non-critical EA is lost, the system continues to operate correctly. For example, losing the icons associated with data files does not generally cause any ill effect other than the inability to show the icon.

A *critical extended attribute* is one which is necessary to the correct operation of OS/2 or of a particular application. EAs should be marked as critical if their loss would cause the system or program to perform incorrectly. For example, a mail program might store mail headers in EAs. The loss of the header from a message would normally render the mail program completely unable to further use that message. This would be unacceptable, so the mail program should mark this EA as critical.

A file has critical extended attributes (EAs) if at least one EA attached to the file is marked as critical.

Marking EAs as critical only prevents programs that do not recognize EAs from losing the EAs from the file. It does not prevent deletion of files by any application.

Applications must be careful how they mark their EAs. If they are too aggressive marking EAs as critical, users might be prevented from accessing files that their application uses. EAs are marked as critical by setting the critical bit. The critical bit is bit 7 of the flags byte of the FEA2 data structure. If this bit is 0, the EA is a non-critical EA. If it is 1, it is a critical EA. The symbolic constant FEA_NEEDEA can be used to indicate a critical EA.

The creator of the EA determines whether it is critical or not.

Programs that do not recognize extended attributes (EAs) are prevented from performing certain operations on files that have critical EAs associated with them. For example, a program that does not recognize EAs is not permitted to perform a non-truncating open on a file with

critical EAs associated with it, because programs cannot be permitted to read the data and ignore the EAs.

Programs that do not recognize EAs are, however, permitted to perform those operations that they can do completely. For example, they can delete files with critical EAs associated with them. Programs that do not recognize EAs are not prevented from accessing files whose EAs are not critical.

Programs that recognize EAs have no restrictions placed on their actions with respect to critical EAs.

Programs that recognize extended attributes must identify themselves to OS/2. This is done by including the NEWFILES declaration in the program's module definition file. The NEWFILES declaration is also how programs indicate that they understand and use long file names.

Searching for Extended Attributes

An application can search for file objects that have specific extended attribute names by calling DosFindFirst twice. The steps involved are:

1. Call DosFindFirst for FileInfoLevel = 2, to get the length of the buffer required to hold the EAOP2 data associated with a matching file object.
2. Call DosFindFirst for FileInfoLevel = 3, to get the EAOP2 data associated with the matching file object.

Supporting Extended Attributes

To support extended attributes, applications should do the following:

1. Fill in the .ASSOCTABLE extended attribute for all major file types that the application recognizes or uses.
2. Fill in the .ICON extended attribute for executable files.
3. Set the .TYPE extended attribute for data files the application creates.
4. Fill in and use the .LONGNAME extended attribute as appropriate.
5. Support .HISTORY and .VERSION.
6. Support the other Standard Extended Attributes as appropriate.

File Management

OS/2 provides a standard set of file system functions. This means that applications can create, open, read, write, copy, and delete files and directories by using the same functions, regardless of which file system is used. When an application calls a file system function, OS/2 passes the request to the dynamic link library that supports the file system. The dynamic link library carries out most file system processing, such as validating file names. If an error occurs, the file system returns the error to OS/2, which then passes it back to the calling application.

The OS/2 file system functions identify files and directories by their names. These functions store or search for the file or directory in the current directory on the current drive unless the name explicitly specifies a different directory and drive. Occasionally, a file system has control functions in addition to the standard file system functions. The control functions are specific to the given file system. An application can call a control function by using DosFSCtl, which directs OS/2 to pass the control-function information to the corresponding Installable File System (IFS).

The following topics are related to the information in this chapter:

- Files systems
- Files names
- Extended attributes
- Pipes
- Device I/O

About Volumes and Drives

OS/2 allows more than one file system on a single storage device. If the device can have more than one partition (or volume), each partition can be initialized as an OS/2 partition and given a valid OS/2 file system. For each volume, OS/2 determines the type of file system the first time the volume is accessed by a function or when the medium in the drive changes. After that, OS/2 manages all input and output to that volume by using the corresponding dynamic link library.

OS/2 uses the volume label and serial number to ensure that the medium in the drive does not change while there are outstanding requests for input and output. Each volume has a volume label and a 32-bit volume serial number, stored in a reserved location in logical sector 0 at the time of formatting. If the volume label and serial number do not match, OS/2 signals the critical error handler to prompt the user to insert the volume that has the specified serial number and label. OS/2 maintains the connection between the medium and the volume label and serial number until all open files on the volume are closed and all search references and cache buffer references are removed. The system redetermines the type of file system and the volume label and serial number only when the medium changes.

OS/2 enables applications to:

- Determine and set the default drive using DosQueryCurrentDisk and DosSetDefaultDisk, respectively.
- Determine and set drive information using DosQueryFSInfo and DosSetFSInfo.
- Determine and set the write verification switch using DosQueryVerify and DosSetVerify. If the write verification switch is on, each time data is written to the disk, the data is verified to ensure it has been recorded correctly. These functions are provided for recording critical data.

About Directories

When an application starts, it inherits the current directory and drive from the application that starts it. An application can determine which directory and drive are current by using DosQueryCurrentDir and DosQueryCurrentDisk. An application can change the current directory and drive of the file system by using DosSetCurrentDir and DosSetDefaultDisk.

When an application creates a new file, the system adds a file entry to the specified directory. Each directory can have any number of entries, up to the physical limit of the disk. An application can create new directories and delete existing directories by using DosCreateDir and DosDeleteDir. Before a directory can be deleted, it must be empty; if there are files or directories in that directory, they must be deleted or moved. An application can delete a file by using DosDelete or move a file to another directory by using DosMove.

Creating a Subdirectory

To create a new subdirectory, an application calls DosCreateDir and specifies a directory path name. If the call is successful, a new subdirectory is created at the end of the path on the specified disk. If no path name is specified, a new subdirectory is created at the end of the current directory for the process. If any subdirectories in the path do not exist, the subdirectory is not created.

Because a subdirectory is a file object, an application also can define an extended attribute for the directory when it is created during this call. See Extended Attributes for more information on extended attributes.

Determining and Changing the Current Directory

Calling DosQueryCurrentDir returns the full path name of the current directory for the specified drive. The string does not begin with a back slash (\) and it ends with a byte containing 00H, the NULL character.

To change the current directory, call DosQueryCurrentDir with the path name of the directory you want to make the current directory.

Deleting a Directory

A subdirectory cannot be removed if it is the current subdirectory or if it contains hidden files or subdirectory entries other than the "." and ".." entries. When these requirements for removal are met, DosDeleteDir can be called with a path name to remove the subdirectory from the disk.

About Files

A file is one or more bytes of data, usually stored on a disk. While the application that creates the file determines the format of the file, the file system determines how the file is stored on the disk and what actions can be performed on it. The file system also stores information about the file in file attributes and extended attributes.

Files are accessed through the file system using file handles. File handles are returned by DosOpen when the file is opened and are used for all subsequent accesses to the file. Files can be be opened, read from, written to, closed, copied, moved, deleted, renamed, and locked. Files can be searched for based on a metacharacter template.

Each open file has a file pointer that indicates the current reading or writing location within the file. The file pointer moves automatically with each read or write operation on the file and can be moved manually by the application.

File Attributes

Each directory entry includes a set of file attributes. File attributes specify whether the directory entry is for a file, a directory, or a volume identifier. The attributes also specify if the file is read-only, hidden, archived, or a system file.

A *read-only* file cannot be opened for writing, nor can it be deleted. A *hidden* file (or directory) cannot be displayed by using an ordinary directory listing. A *system* file is excluded from normal directory searches. The *archived attribute* is for use by special purpose applications to mark a file that has been changed since the last time the file was examined.

An application can retrieve and set the file attributes by using DosQueryFileInfo and DosSetFileInfo.

File Handles

OS/2 identifies each open file by assigning it a file handle when the application opens or creates the file. The file handle is a unique 32-bit integer. The application can use the handle in functions that read from and write to the file, depending on how the file was opened. The application can continue to use the handle until the file is closed.

The default maximum number of file handles for a process is 50. An application can change this maximum by using DosSetMaxFH. When this call is made, all currently open file handles are preserved.

In the past, the maximum number of file handles was 20. If you previously had code that increased the maximum file handles from 20 to less than 50, you can now remove this code.

When an application starts, it inherits all open file handles from the process that starts it. If the system's command processor starts an application, file handles 0, 1, and 2 represent the standard input, standard output, and standard error files. The standard input file is the keyboard; the standard output and standard error files are the screen. An application can read from the standard input file and write to the standard output and standard error files immediately; it does not have to open the files first.

An application can create a duplicate file handle for an open file by using DosDupHandle. A duplicate handle is identical to the original handle. Typically, duplicate handles are used to redirect the standard input and standard output files. For example, an application can open a disk file and duplicate the disk-file handle as handle 0. Thereafter, an application reading from the standard input file (handle 0) takes data from the disk file, not from the keyboard.

When devices and pipes are accessed through the file system functions (using DosOpen, DosRead, and so on), the devices and pipes are treated as files and are identified using file handles. The standard file handles can be redirected to a device or pipe.

File Pointer

Every open file has a file pointer that specifies the next byte to be read or the location to receive the next byte that is written. When a file is first opened, the system places the file pointer at the beginning of the file. As each byte is read or written, OS/2 advances the pointer.

An application can also move the pointer by using DosSetFilePtr. When the pointer reaches the end of the file and the application attempts to read from the file, no bytes are read and no error occurs. Thus, reading 0 bytes without an error means the program has reached the end of the file.

When an application writes to a disk file, the data being written is usually collected in an internal buffer. OS/2 writes to the disk only when the amount of data equals (or is a multiple of) the sector size of the disk. If there is data in the internal buffer when the file is closed, the system automatically writes the data to the disk before closing the file. An application can also flush the buffer (that is, write the contents of the buffer to the disk) by using DosResetBuffer.

Copying Files

DosCopy is used to copy files and subdirectories. Metacharacters (global file name characters) are not allowed, so only individual files or entire subdirectories can be copied with this function. The source and target can be on different drives.

When the source specified is a subdirectory and an I/O error occurs, the file being copied from the source directory to the target directory at the time of the error will be deleted from the target directory and DosCopy will be terminated.

When a file is being copied and an I/O error occurs,

- if the source file name is that of a file to be replaced, the file is deleted from the target path.
- if the source file name is that of a file to be appended, the target file is resized to its original size.

DosCopy will copy the attributes of the source file, such as date of creation, and time of creation to the target file. Additionally, DosCopy will copy the extended attributes of the source file when creating a new subdirectory or a new file, or when replacing an existing file.

Moving Files

DosMove is used to move a file from one subdirectory to another on the same drive. In the process of moving the file, its name can be changed. Metacharacters (global file name characters) are not permitted.

Deleting Files

Calling DosDelete removes the directory entry associated with a file name. Metacharacters (global file name characters) are not permitted.

Files whose read-only attribute is set cannot be deleted.

Changing File Sizes

DosSetFileSize is used to extend or truncate the size of a file that is open for Read/Write or Write-Only access.

When a file is extended, for example to reserve disk space, the value of the additional bytes allocated by the system is undefined.

Locking and Unlocking File Regions

Because OS/2 permits more than one application to open a file and write to it, it is important that the applications not write over each other's

work. An application can protect against this problem by temporarily locking a region in a file.

DosSetFileLocks provides a simple mechanism that temporarily prevents access by other processes to regions within a file. DosSetFileLocks specifies a range of bytes in the file that is locked by an application and that can be accessed only by the application locking the region. The application uses the same function to free the locked region.

Locking and unlocking regions in a file enables sharing processes to coordinate their efforts. A process can lock a region in a file so the process can read and update the file region. A sharing process that attempts to lock the region before the other process finishes its update and unlocks the region receives an error code. When a lock is unsuccessful because a lock is already in place, ERROR_LOCK_VIOLATION is returned.

A lock that extends beyond end-of-file is not considered an error. However, a locked region cannot overlap another locked region, nor can it encompass a locked region. Any such conflicting locks must be cleared before a region can be locked.

When a file handle is duplicated, the duplicate handle has access to any locked regions currently being accessed by the original handle. Although a child process created with DosExecPgm can inherit the file handles of its parent process, it does not inherit access to locked regions in effect for a file handle unless the file handle is duplicated and passed to it.

Note: File locks are intended to be in effect for only short periods of time.

When a file with locks is closed, the locks are released in no defined order.

Searching for Files

An application can use DosFindFirst, DosFindNext, and DosFindClose to search the current directory for all file names that match a given pattern.

The pattern must be an OS/2 file name and can include metacharacters (global file name characters). The wildcard characters are the question mark (?) and the asterisk (*). The question mark matches any single character; the asterisk matches any combination of characters. For example, the pattern "A*" matches the names "ABC", "A23", and "ABCD", but the pattern "A?C" matches only the name "ABC".

Determining the Maximum Path Length

An application that recognizes long file names might be run on systems with or without installable file systems. Additionally, the maximum length of a file name might vary from one installable file system to another. So that an application can properly handle this variable condition (and, for example, allocate large enough buffers to hold file names), the application should call DosQuerySysInfo to determine the maximum path length supported by the file system.

Make this call before calling file system functions that require full path names.

Specifying an Extended LIBPATH

The LIBPATH, which is set in CONFIG.SYS, specifies a search path which OS/2 uses when searching for dynamic link libraries (DLLs). The LIBPATH is set during system startup and cannot be changed dynamically.

OS/2 provides the capability of specifying extensions to the LIBPATH. An *Extended LIBPATH* is a path which is searched in conjunction with the system LIBPATH, but which can be changed dynamically, either by the user from the command line, or by an application.

There are two Extended LIBPATHs: *BeginLIBPATH*, which is searched before the system LIBPATH, and *EndLIBPATH*, which is searched after the system LIBPATH.

Extended LIBPATHs are ASCIIZ strings which are formatted in the same manner as the system LIBPATH, that is, they contains a list of subdirectory paths separated by semicolons. The maximum size for each Extended LIBPATH is 1024 characters.

Applications can use DosSetExtLIBPATH to set the Extended LIBPATHs. Likewise, they can use DosQueryExtLIBPATH to query the value of either of the Extended LIBPATHs. A flag parameter for the function specifies whether the BeginLIBPATH or the EndLIBPATH is being set or queried. The flag allows two values: BEGIN_LIBPATH (1) which will set or query BeginLIBPATH, or END_LIBPATH (2) which will set or query EndLIBPATH.

You can call DosSetExtLIBPATH as often as you want. The new Extended LIBPATH that you pass in will replace the current Extended LIBPATH.

You can specify the current Extended LIBPATH as part of the argument by using the **%** symbol before and after the Extended LIBPATH variable name, that is, as *%BeginLIBPATH%* or *%EndLIBPATH%*. For example, suppose you set BeginLIBPATH to "D:\MYDLLS", then later you want to add "D:\TOMSDLLS" before the existing BeginLIBPATH, and "D:\JOESDLLS" after the existing BeginLIBPATH (that is, you want to change BeginLIBPATH to "D:\TOMSDLLS;D:\MYDLLS;D:\JOESDLLS"). You could accomplish this by using "D:\TOMSDLLS;%BeginLIBPATH%;D:\JOESDLLS" as the argument for DosSetExtLIBPATH.

The string argument can be up to 1024 characters long. However, if the resulting Extended LIBPATH is longer than 1024 characters, the function will return an error.

Devices

OS/2 enables you to access peripheral devices using file system commands, and treat those devices as file streams of data. These devices include:

Character devices	COM, clock, console (keyboard and screen), screen, keyboard, printer, null, pointer, and mouse devices.
Standard I/O devices	Character devices automatically installed by OS/2 and recognized by the file system as the standard input, standard output, and standard error devices.
Pseudocharacter devices	An application can attach a device name to a file system and use the file system as a pseudocharacter device (also called a single-file device). Attaching a device name to a file system allows an application to open the device associated with the file system as if it were a character device (for example, a serial port) and read from and write to the device by using DosRead and DosWrite. In addition, an application can use DosSetFilePtr and DosSetFileLocks with a pseudocharacter device. Also, pseudocharacter devices can be redirected. A file system that can be attached to a pseudocharacter device is typically associated with a single disk file or with a special storage device, such as a tape drive. The file system establishes a connection with the device and transfers requests and data between OS/2 and the device. The user perceives this file as a device name for a nonexistent device. This file is seen as a character device because the current drive and directory have no effect on the name. A pseudocharacter device name is an ASCII string with the name of an OS/2 file in the form: `\DEV\DEV_NAME` The "\dev\" is a required part of the name, but there is no actual subdirectory named "\dev\". This just lets OS/2 know that it is a pseudocharacter device name.
Logical file devices	The hard disk or floppy disk drives, or the partitions on the hard disk or floppy disk drives.

Using File Commands

Files are accessed through the file system using file handles. File handles are returned by DosOpen when the file is opened and are used for all subsequent accesses to the file. Files can be be created, opened, read from, written to, closed, copied, moved, deleted, renamed, and locked. Files can be searched for based on a metacharacter pattern template.

Each open file has a file pointer that indicates the current reading or writing location within the file. The file pointer moves automatically with each read or write operation on the file and can be moved manually by the application.

The standard file handles-standard input, standard output, and standard error-provide redirectable input and output to applications. The standard file handles can be used to read input from the keyboard and write output to the display. Alternatively, they can be redirected to read input from and write output to a file. To an application reading and writing the standard file handles, there is no difference between the two.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Creating Files

DosOpen is used to create files, which are then read from or written to. To create a file, specify FILE_CREATE as the sixth argument in the call to the function. DosOpen then creates the file, if it does not already exist. If the file already exists, the function returns the error value FILE_EXISTED.

The following code fragment shows how to use DosOpen to create the file NEWFILE.TXT:

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>

HFILE   hf;
ULONG   ulAction;
APIRET  ulrc;

ulrc = DosOpen("NEWFILE.TXT",       /* Name of file to create and open */
               &hf,                 /* Address of file handle          */
               &ulAction,           /* Action taken                    */
               0,                   /* Size of new file                */
               FILE_NORMAL,         /* File attributes                 */
               FILE_CREATE,         /* Creates the file                */
               OPEN_ACCESS_WRITEONLY |
               OPEN_SHARE_DENYNONE,
               (PEAOP2) NULL);      /* No extended attributes          */
```

In this example, DosOpen creates the file and opens it for writing only. Note that the sharing method chosen permits other processes to open the file for any access. The new file is empty (contains no data).

When you use DosOpen to create (or replace) a file, you must specify the attributes the new file is to have. In the preceding example, this value is FILE_NORMAL, so the file is created as a normal file. Other possible file attributes include read-only and hidden, which correspond to FILE_READONLY and FILE_HIDDEN, respectively. The possible file attributes are:

File Attribute	Defined Constant
Normal file	FILE_NORMAL
Read-only file	FILE_READONLY
Hidden file	FILE_HIDDEN
System file	FILE_SYSTEM
Subdirectory	FILE_DIRECTORY
Archived file	FILE_ARCHIVED.

The file attribute affects how other processes access the file. For example, if the file is read-only, no process can open the file for writing. There is one exception-the process that creates the read-only file can write to it immediately after creating it. After closing the file, however, the process cannot reopen it for writing.

If you are creating a new file object (a new file or a replacement for an existing one), you must specify the size of the new file in bytes. For example, if you specify 256, the file size is 256 bytes. However, these 256 bytes are undefined. If the file being opened already exists, the file size parameter is ignored. It is up to the application to write valid data to the file. No matter what size you specify, the file pointer is set to point to the beginning of the file so a subsequent call to DosWrite starts writing data at the beginning of the file.

Extended attributes can be defined by an application when a file object is created. An application can define an extended attribute for a file when the file is created during a DosOpen call.

Applications can also control access to specific regions within a file by calling DosSetFileLocks.

Opening Files

Before performing input or output operations on a file, you must open the file and obtain a file handle. You obtain a file handle by using

DosOpen. This function opens the specified file and returns a file handle for it. DosOpen can also be used to create new files.

DosOpen establishes a connection betwee n a file object and an application. This connection is in the form of a 32-bit identifier called a *file handle* , which is used to refer to the file object and any information associated with it. DosOpen returns a handle that is used in other file system calls to gain access to the object. The file object can be a new file, an existing file, or a replacement for an existing file. It can also be a character device, a block device, or the client end of a named pipe. The type of object is determined by the file name you pass to DosOpen.

Note: If the object is a named pipe, it must be in a listening state for DosOpen to be successful.

The following code fragment shows the use of DosOpen to:

- Open the existing file SAMPLE.TXT for reading
- Put the file handle into the *hf* variable

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>

HFILE   hf;
ULONG   ulAction;
APIRET  ulrc;

ulrc = DosOpen("SAMPLE.TXT",       /* Name of file to open      */
               &hf,                /* Address of file handle    */
               &ulAction,          /* Action taken              */
               0,                  /* Size of file              */
               FILE_NORMAL,        /* File attribute            */
               FILE_OPEN,          /* Open the file             */
               OPEN_ACCESS_READONLY |
               OPEN_SHARE_DENYNONE,
               (PEAOP2) NULL);     /* Extended attribute buffer */
```

If DosOpen successfully opens the file, it copies the file handle to the *hf* variable and copies a value to the *ulAction* variable indicating what action was taken (for example, FILE_EXISTED for "existing file opened"). A file size is not needed to open an existing file, so the fourth argument is 0. The fifth argument, FILE_NORMAL, specifies the normal file attribute. The sixth argument, FILE_OPEN, directs DosOpen to open the file if it exists or return an error if it does not exist. The seventh argument directs DosOpen to open the file for reading only and enables other applications to open the file even while the current application has it open. The final argument is a pointer to a structure containing information on extended attributes. If the file has no extended attributes, this argument must be NULL.

DosOpen returns 0 if it successfully op ens the file. Applications use the file handle in subsequent functions to read data from the file or to check the status and other file characteristics. If DosOpen fails to open the file, it returns an error value.

When you open a file you must specify whether you want to read from the file, write to it, or both read and write. Also, since more than one application might attempt to open the same file, you must also specify whether you want to allow other processes to have access to the file while you have it open. A file that is shared can be shared for reading only, writing only, or reading and writing. A file that is not shared cannot be opened by another application (or more than once by the first application) until it has been closed by the first application.

An application defines its file access rights (that is, I/O it needs to perform on a file) by setting the appropriate file access mode field in the file open-mode parameter. An application accesses a file as:

- Read-only
- Write-only
- Read/write

An application defines what I/O operations other processes can perform on a file by setting the appropriate sharing mode field in the *OpenMode* parameter. Other processes are granted:

- Deny read/write access
- Deny write access
- Deny read access
- Deny neither read or write access (deny none)

File sharing requires cooperation between the sharing processes. For example, if process A calls DosOpen to open a file with Deny Write sharing and process B calls DosOpen to open the same file with Read/Write access, the DosOpen call made by process B fails.

You indicate whether or not you want to permit another application to access your file by combining an OPEN_ACCESS_ value and an OPEN_SHARE_ value from the following list:

File Access and Sharing Rights

Value	Meaning
OPEN_ACCESS_READONLY	Open a file for reading.

OPEN_ACCESS_WRITEONLY	Open a file for writing.
OPEN_ACCESS_READWRITE	Open a file for reading and writing.
OPEN_SHARE_DENYREADWRITE	Open a file for exclusive use, denying other processes read and write access.
OPEN_SHARE_DENYWRITE	Deny other processes write access to a file.
OPEN_SHARE_DENYREAD	Deny other processes read access to a file.
OPEN_SHARE_DENYNONE	Open a file with no sharing restrictions, granting read and write access to all processes.

In general, you can combine any access method (read, write, or read and write) with any sharing method (deny reading, deny writing, deny reading and writing, or grant any access). Some combinations have to be handled carefully, however, such as opening a file for writing without denying other processes access to it.

Note: For named pipes, the access and sharing modes must be consistent with those specified by DosCreateNPipe.

Other characteristics of the file handle that can be set:

File Handle Characteristics

Flag	Purpose
Inheritance	Handle is inherited by a child process created with DosExecPgm, or is private to the calling process.
Write-Through	Actual I/O for synchronous writes is not guaranteed as complete or is guaranteed as complete before the write returns.
Fail-Errors	Media errors are reported by the system critical error handler, or by the application.
DASD	The file name parameter is the name of a file or device opened in the normal way, or a drive specification for a fixed disk or diskette drive. The DASD flag can be set for direct access to an entire disk or diskette volume, independent of the file system. When the DASD flag is set, the handle returned by DosOpen represents the logical volume as a single file. To block other processes from accessing the logical volume, a DosDevIOCtl Category 8, Function 0 call should be made using the handle returned by DosOpen. The DASD flag should be set only by systems programs, not by applications.
Cache	The file system caches or does not cache data for I/O operations on the file. This flag is advisory only.

See the DosOpen material in the *Control Program Programming Reference* for details of these characteristics.

After an object has been opened, its file handle state flags can be queried by calling DosQueryFHState and reset by calling DosSetFHState. See Determining and Setting the State of a File or Device Handle for information on determining the state of a file handle.

When a child process inherits a file handle, it also inherits the sharing and access rights of the file.

You cannot use metacharacters (global file name characters; * and ?) in file names you supply to DosOpen.

Reading from Files

Once you open a file or have a file handle, you can read from and write to the file by using DosRead and DosWrite.

DosRead is called with a handle (obtained with DosOpen) for a file, pipe, or device. DosRead copies a specified number of bytes (up to the end of the file) from the file object to the buffer you specify. OS/2 returns, in a parameter, the number of bytes actually read (which might not be the same as the number of bytes requested because the end of the file might have been reached).

To read from a file, you must open it for reading or for reading and writing.

The following code fragment shows how to open the file named SAMPLE.TXT and read the first 512 bytes from it:

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>

#define BUF_SIZE 512

HFILE   hf;
ULONG   ulAction;
BYTE    abBuffer[BUF_SIZE];
ULONG   cbRead;
APIRET  ulrc;

ulrc = DosOpen("SAMPLE.TXT",
               &hf,
               &ulAction,
               0,
               FILE_NORMAL,
               FILE_OPEN,
               OPEN_ACCESS_READONLY |
               OPEN_SHARE_DENYNONE,
               (PEAOP2) NULL);

if (!ulrc) {
    DosRead(hf,
            abBuffer,
            sizeof(abBuffer),
            &cbRead);

    DosClose(hf);
}
```

If the file does not have 512 bytes, DosRead reads to the end of the file and puts the number of bytes read in the *cbRead* variable. If the file pointer is already positioned at the end of the file when DosRead is called, the function puts a 0 in the *cbRead* variable.

Writing to Files

Once you open a file or have a file handle, you can read from and write to the file by using DosRead and DosWrite.

DosWrite copies bytes from a buffer you specify to a file, device, or pipe.

Calling DosWrite with a handle for a file, pipe, or device transfers the number of bytes specified from a buffer location to the object. The system returns, in a parameter, the number of bytes actually written (which in the case of a disk file might not be the same as the number requested because of insufficient disk space).

To write to a file, you must first open it for writing or for reading and writing.

The following code fragment shows how to open the file SAMPLE.TXT again and write 512 bytes to it:

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>

#define BUF_SIZE 512

HFILE   hf;
ULONG   ulAction;
BYTE    abBuffer[BUF_SIZE];
ULONG   cbWritten;
APIRET  ulrc;

ulrc = DosOpen("SAMPLE.TXT",
               &hf,
               &ulAction,
               0,
               FILE_NORMAL,
               FILE_CREATE,
               OPEN_ACCESS_WRITEONLY |
               OPEN_SHARE_DENYWRITE,
               (PEAOP2) NULL);

if (!ulrc) {
    DosWrite(hf,
             abBuffer,
             sizeof(abBuffer),
             &cbWritten);

    DosClose(hf);
}
```

DosWrite writes the contents of the buffer to the file. If it fails to write 512 bytes (for example, if the disk is full), the function puts the number of bytes written in the *cbWritten* variable. The data is read and written exactly as given; the function does not format the data-that is, they do not convert binary data to decimal strings, or vice versa.

The Write-Through Flag
If an application requires data to be written in a specific order, setting the *Write-Through flag* to 1 guarantees that actual I/O for a synchronous write is completed before the DosWrite returns. If this flag has been set with DosOpen for buffered I/O, and multiple synchronous writes are performed, the system cannot guarantee the actual order in which the data is written. For details on changing the state of the *Write-Through flag* , see Determining and Setting the State of a File or Device Handle.

Moving the File Pointer

Every disk file has a corresponding file pointer that marks the current location in the file. The current location is the byte in the file that will be read from or written to on the next call to DosRead or DosWrite. Usually, the file pointer is at the beginning of the file when you first open or create the file, and it advances as you read or write to the file. You can, however, change the position of the file pointer at any time by using DosSetFilePtr.

DosSetFilePtr moves the file pointer a specified offset from a given position. You can move the pointer from the beginning of the file, from the end, or from the current position.

The following code fragment shows how to move the pointer 200 bytes from the end of the file:

```
#define INCL_DOSFILEMGR   /* File system values     */
#include <os2.h>

#define HF_STDOUT 1       /* Standard output handle */
#define BUF_SIZE 255

BYTE abBuf[BUF_SIZE];
HFILE hf;
ULONG ulRead,
      ulWritten,
      ulAction,
```

```
      ulNewPtr;

DosOpen("SAMPLE.TXT",
        &hf,
        &ulAction,
        0,
        FILE_NORMAL,
        FILE_OPEN,
        OPEN_ACCESS_READONLY |
        OPEN_SHARE_DENYNONE,
        (PEAOP2) NULL);

DosSetFilePtr(hf,
              -200,
              FILE_END,
              &ulNewPtr);

DosRead(hf,
        &abBuf,
        sizeof(abBuf),
        &ulRead);

DosWrite(HF_STDOUT,
         abBuf,
         ulRead,
         &ulWritten);
```

In this example, DosSetFilePtr moves the file pointer to the 200th byte from the end of the file. If the file is not that long, the function moves the pointer to the first byte in the file and returns the actual position (relative to the end of the file) in the *ulNewPtr* variable.

You can move the file pointer for disk files only. You cannot use DosSetFilePtr on devices, despite using other file functions (DosOpen, DosRead) to access a device.

If a file is read-only, write operations to the file will not be performed.

Moving the pointer from the end of the file can be used to determine the size of the file.

Closing Files

You can close a file by using DosClose. Since each application has a limited number of file handles that can be open at one time, it is a good practice to close a file after using it.

To do so, supply the file handle in DosClose, as shown in the following code fragment:

```
#define INCL_DOSFILEMGR   /* File system values */
#include <os2.h>

#define BUF_SIZE 80

HFILE   hf;
ULONG   ulAction;
BYTE    abBuffer[BUF_SIZE];
ULONG   ulRead;
APIRET  ulrc;

ulrc = DosOpen("SAMPLE.TXT",
               &hf,
               &ulAction,
               0,
               FILE_NORMAL,
               FILE_OPEN,
               OPEN_ACCESS_READONLY |
               OPEN_SHARE_DENYNONE,
               (PEAOP2) NULL);

if (!ulrc) {
    DosRead(hf,
            abBuffer,
            sizeof(abBuffer),
            &ulRead);

    DosClose(hf);
```

```
}
```

If you open a file for writing, DosClose directs the system to flush the file buffer-that is, to write any existing data in OS/2's intermediate file buffer to the disk or device. The system keeps these intermediate file buffers to make file input and output more efficient. For example, it saves data from previous calls to DosWrite until a certain number of bytes are in the buffer then writes the contents of the buffer to the disk.

DosClose also closes the handle to the file (or pipe, or device). If one or more additional handles to a file have been created with DosDupHandle , the internal buffers for the file are not written to disk, and its directory entry is not updated, until DosClose has been called for all the handles.

Creating Duplicate File or Device Handles

DosDupHandle enables a process to create a duplicate handle for an open file, pipe, or device.

The value for the old-file-handle parameter is the handle of an open file, a pipe, or a device. The valid values for the new-file-handle parameter include FFFFH, 0000H (standard input), 0001H (standard output), and 0002H (standard error). Any value other than FFFFH is assumed to be the value of the new file handle.

A value of FFFFH causes the system to allocate a new file handle and send it to this location. If the value specified for the new-file-handle parameter is that of a currently open file, the file handle is closed before it is redefined.

An agreed upon value for a duplicate file handle can be established between a parent process and a child process. Avoid choosing arbitrary values for the new file handle.

The duplicate handle acquires the characteristics of the original. If you move the read/write pointer of the original file handle, for example by calling DosRead, DosWrite, or DosSetFilePtr , the pointer of the duplicate handle is also moved. If the original handle has access to regions in a file that have been locked by DosSetFileLocks , the duplicate also has access.

If inheritance was indicated when a file was opened with DosOpen, a parent process can create a duplicate handle for the file and pass it to a child process by means of shared memory. This permits the child to close the duplicate handle without affecting the file handle of the parent.

Because a parent process controls the meanings for standard I/O done by any child process it creates, the parent can use DosDupHandle to redefine unnamed pipe handles as standard I/O handles to communicate with a child. The steps involved are:

- The parent process creates two pipes and duplicates the read handle of one pipe as 0000 and the write handle of the other pipe as 0001.
- When the child process performs standard I/O, instead of reading from the keyboard and writing to the display, it reads from and writes to the pipes created by its parent.
- As the owner of the pipe, the parent process uses its read and write handles to write to the pipe defined to the child as standard input and read from the pipe defined to the child as standard output.

Determining and Setting the State of a File or Device Handle

After a file has been opened, the file handle state flags set with a DosOpen can be queried and reset by calling DosQueryFHState and DosSetFHState . The handle returned by DosSetFHState is used for subsequent input and output to the file.

The following code fragment calls DosSetFHState to set the File Write-Through flag for an opened file. Writes to the file may go through the file system buffer, but the sectors are to be written before any synchronous write call returns. DosQueryFHState is called first to obtain the file handle state bits. Assume that the appropriate file handle has been placed into *FileHandle* already.

```
#define INCL_DOSFILEMGR         /* File system values */
#include <os2.h>
#include <stdio.h>

HFILE   hfFileHandle;          /* File handle        */
ULONG   ulFileHandleState;     /* File handle state  */
APIRET  ulrc;                  /* Return code        */

ulrc = DosQueryFHState(hfFileHandle,
```

```
                        &FileHandleState);

    if (ulrc != 0) {
        printf("DosQueryFHState error: return code = %ld",
               ulrc);
        return;
    }

    ulFileHandleState |= OPEN_FLAGS_WRITE_THROUGH;

    ulrc = DosSetFHState(hfFileHandle,
                         ulFileHandleState);

    if (ulrc != 0) {
        printf("DosSetFHState error: return code = %ld",
               ulrc);
        return;
    }
```

Here are two scenarios involving the use of this function.

- An application requires that data be written in a specific order. To guarantee the order of the data written, it must perform separate synchronous write operations. The application can call DosSetFHState to set the Write-Through flag for the file. This action does not affect any previous asynchronous writes, whose data can remain in the buffers.
- An application cannot handle a certain critical error situation. DosSetFHState can be called to reset critica l error handling as OS/2's responsibility. The I/O function that caused the critical error must be called again so the error can recur, causing control to be passed to OS/2. In the case where asynchronous I/O is being done, the precise time the results of this function will be available to the application is unpredictable.

Determining the Handle Type

DosQueryHType enables an application to determine whether a handle is to a file, a pipe, or a device. This function can be used when a file-oriented application needs to modify its behavior, depending on the source of its input. For example, CMD.EXE suppresses writing prompts when its input is from a disk file.

The following code fragment determines whether a given file handle refers to a file or a device. Assume that the desired file handle has been placed into *FileHandle* already.

```
    #define INCL_DOSFILEMGR   /* File system values                 */
    #include <os2.h>
    #include <stdio.h>

    HFILE   hfFileHandle;    /* File handle                          */
    ULONG   ulHandType;      /* Handle type (returned)               */
    ULONG   ulFlagWord;      /* Device driver attribute (returned)   */
    APIRET  ulrc;            /* Return code                          */

    ulrc = DosQueryHType(hfFileHandle,
                         &ulHandType,
                         &ulFlagWord);

    if (ulrc != 0) {
        printf("DosQueryHType error: return code = %ld",
               ulrc);
        return;
    }
```

In the preceding example, DosQueryHType returns a value that characterizes the type of file handle, and the associated device driver attribute word, if the handle type indicates that the file handle is associated with a local character device.

Searching for Files

You can locate files with names that match a given pattern by using metacharacters in DosFindFirst and DosFindNext .

DosFindFirst searches the current directory and locates the first file name that matches the given pattern. DosFindNext locates the next matching file name and continues to find additional matches on each subsequent call until all matching names are found. The functions copy the file statistics on each file located to a data structure that you supply. The file information returned by a search includes file dates and times, length of data in the file, file size, file attributes, and file name.

To find all files that match the file specification, call DosFindNext repeatedly until the message ERROR_NO_MORE_FILES is returned. Then call DosFindClose to close the directory handle.

The following code fragment shows how to find all file names that have the extension ".C":

```
#define INCL_DOSFILEMGR   /* File system values */
#include <os2.h>

HDIR         hdHdir;
ULONG        ulFilenames;
FILEFINDBUF3 ffbFindBuf;
APIRET       ulrc;

ulFilenames = 1;
hdHdir = HDIR_SYSTEM;

ulrc = DosFindFirst("*.C",
                    &hdHdir,            /* Directory handle                    */
                    FILE_NORMAL,        /* File attribute to look for          */
                    &ffbFindBuf,        /* Result buffer                       */
                    sizeof(ffbFindBuf), /* Size of result buffer               */
                    &ulFilenames,       /* Number of matching names to look for */
                    FIL_STANDARD);      /* Standard level of information       */

if (!ulrc) {
    do {
        .
        .     /* Use file name in findbuf.achName */
        .
        ulrc = DosFindNext(hdHdir,
                           &ffbFindBuf,
                           sizeof(ffbFindBuf),
                           &ulFilenames);

    } while (!ulrc);
}
DosFindClose(hdHdir);
```

In this example, DosFindNext continues to retrieve matching file names until it returns an error value (it returns ERROR_NO_MORE_FILES when it cannot find any more matching files).

To keep track of which files have already been found, both functions use the directory handle, *hdir* .

This directory handle must be set to HDIR_SYSTEM or HDIR_CREATE before the call to DosFindFirst. HDIR_SYSTEM (00000001H) tells OS/2 to use the system handle for standard output, which is always available to the requesting process. HDIR_CREATE (FFFFFFFFH) tells OS/2 to allocate a new, unused handle.

The directory handle returned by DosFindFirst must be used in subsequent calls to DosFindNext ; it identifies for DosFindNext the name of the file being sought and the current position in the directory.

An attribute parameter for DosFindFirst allows hidden and system files, as well as normal files, to be included in searches.

After locating the files you need, use DosFindClose to close the directory handle. This ensures that when you search for the same files again, you will start at the beginning of the directory. After DosFindClose is called, a subsequent DosFindNext fails.

Searching Paths for Files

DosSearchPath searches directory paths for t he name of a file object. The file specification can include metacharacters (global file name characters).

The path string used in the search consists of directory paths separated by semicolons. The caller can supply the path string, or it can supply the name of an environment variable whose value is the path string to be searched. The caller can request that the current working directory

be searched before the path string is searched.

If the caller specifies an environment variable, DosSearchPath uses DosScanEnv to find the path string. DosScanEnv searches the environment segme nt for an environment variable name; for example, DPATH. The result pointer points to the string that is the value of the environment variable. The call to DosScanEnv can be made direct ly by the application, or it can be invoked by DosSearchPath.

If the file is found, its full path name is returned, with metacharacters left in place. The results might not be meaningful if a buffer overflow occurs.

As an example, assume that a string such as the following exists in the environment:

```
DPATH=C:\SYSDIR;C:\INIT
```

The following code fragment illustrates a method for searching directory paths to find a file.

```
#define INCL_DOSFILEMGR   /* File system values */
#define INCL_DOSMISC
#include <os2.h>
#include <stdio.h>

#define ResultBufLen 255

INT main(VOID)
{
    PSZ  pszPathRef="";
    UCHAR achResultBuffer[ResultBufLen};
    PSZ   pszFile="OS2.INI";

    DosScanEnv("DPATH",
               &pszPathRef);

    DosSearchPath(0,                   /* Path source bit=0 */
                  pszPathRef,
                  pszFile,
                  achResultBuffer,
                  ResultBufLen);

    printf("Result 1: %s\n",
           achResultBuffer);

    DosSearchPath(2,                   /* Path source bit=1 */
                  (PSZ)"DPATH",
                  pszFile,
                  achResultBuffer,
                  ResultBufLen);

    printf("Result 2: %s\n",
           achResultBuffer);

    return;

}
```

Specifying Extended LIBPATHs

An *Extended LIBPATH* is a path which is searched in conjunction with the system LIBPATH, but which can be changed dynamically, either by the user from the command line, or by an application. There are two Extended LIBPATHs: *BeginLIBPATH*, which is searched before the system LIBPATH, and *EndLIBPATH*, which is searched after the system LIBPATH.

Applications can use DosSetExtLIBPATH to set the Extended LIBPATHs. Likewise, they can use DosQueryExtLIBPATH to query the value of either of the Extended LIBPATHs. A flag parameter for the function specifies whether the BeginLIBPATH or the EndLIBPATH is being set or queried. The flag allows two values: BEGIN_LIBPATH (1) which will set or query BeginLIBPATH, or END_LIBPATH (2) which will set or query EndLIBPATH.

Extended LIBPATHs are ASCIIZ strings which are formatted in the same manner as the system LIBPATH, that is, they contains a list of subdirectory paths separated by semicolons. The string argument can be up to 1024 characters and will return an error if longer.

The following example updates the path to be searched before the system LIBPATH, then queries and displays the value.

```
#define INCL_DOSMISC
#define INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>

UCHAR   uchBeginLIBPATH[1024] = "";       /* Begin LIBPATH value returned */
APIRET  ulrc                  = NO_ERROR; /* Return code                  */

   ulrc = DosSetExtLIBPATH("C:\\TOOL_X\\VERS_20\\DLL",
                           BEGIN_LIBPATH);    /* Add to beginning LIBPATH */

   if (ulrc != NO_ERROR) {
       printf("DosSetExtLIBPATH error: return code = %u\n",
              ulrc);
       return 1;
   }

   ulrc = DosQueryExtLIBPATH(uchBeginLIBPATH,
                             BEGIN_LIBPATH);   /* Query the BeginLIBPATH */

   if (ulrc != NO_ERROR) {
       printf("DosQueryExtLIBPATH error: return code = %u\n",
              ulrc);
       return 1;
   }

   printf(" BeginLIBPATH = %s\n",
          uchBeginLIBPATH);
```

Standard File Handles

Every application, when it first starts, has three input and output file handles available to use. These file handles, called the standard input, standard output, and standard error files, enable the application to read input from the keyboard and display output on the screen without opening or preparing the keyboard or screen.

Standard Input, Output, and Error File Handles

As OS/2 starts an application, it automatically opens the three standard files and makes the file handles-numbered 0, 1, and 2-available to the application. Applications can read from and write to the standard files as soon as they start.

Standard Input
File handle 0 is the standard input file. This handle can be used to read characters from the keyboard with DosRead. The function reads the specified number of characters unless the user types a *turnaround character* -that is, a character that marks the end of a line (the default turnaround character is a carriage-return/linefeed character pair).

As DosRead reads the characters, it copies them to the buffer you have supplied, as shown in the code fragment below.

In this example, DosRead copies the number of characters read from standard input to to variable *cbRead* . DosRead also copies the turnaround character, or characters, to the buffer If the function reads fewer than 80 characters, the turnaround character is the last one in the buffer.

Standard Output
File handle 1 is the standard output file. This handle can be used to write characters on the screen with DosWrite. The function writes the characters in the given buffer to the current line. If you want to start a new line, you must place the current turnaround character in the buffer.

The code fragment below:

- Displays a prompt
- Reads a string
- Displays the string

```
#define INCL_DOSFILEMGR   /* File system values     */
#include <os2.h>

#define HF_STDIN  0       /* Standard input handle  */
#define HF_STDOUT 1       /* Standard output handle */
#define BUF_SIZE 80

ULONG ulWritten,
      ulRead;

BYTE abBuffer[BUF_SIZE];

static UCHAR ucEnterName[] = "Enter a name: ";

DosWrite(HF_STDOUT,
         ucEnterName,
         sizeof(ucEnterName),
         &ulWritten);

DosRead(HF_STDIN,
        abBuffer,
        sizeof(abBuffer),
        &ulRead);

DosWrite(HF_STDOUT,
         abBuffer,
         ulRead,
         &ulWritten);
```

Standard Error
File handle 2 is the standard error file. This handle, like the standard output handle, enables applications to write characters on the screen. Most applications use the standard error file to display error messages, enabling the application to redirect standard output to a file without also redirecting error messages to the file.

Redirecting Standard File Handles

The standard input, standard output, and standard error files are usually the keyboard and screen, but not always. For example, if you redirect standard output by using the greater-than (>) redirection symbol on the application's command line, all data written to the standard output file goes to the given file.

The following command line redirects standard output to the file SAMPLE.TXT and redirects error messages to the file SAMPLE.ERR:

```
type startup.cmd >sample.txt 2>sample.err
```

When a standard file is redirected, its handle is still available but corresponds to the given disk file instead of to the keyboard or screen. You can still use DosRead and DosWrite to read from and write to the files.

You can use DosDupHandle to redirect a standard file from inside your application. If you duplicate the standard input file handle, your application reads from the specified file rather than from the keyboard. Duplicating the standard output file handle causes output to be directed to a file or device instead of to the standard output device.

The following code fragment shows how to use the standard input handle to read from a file:

```
#define INCL_DOSFILEMGR   /* File system values     */
#include <os2.h>

#define HF_STDIN  0       /* Standard input handle  */
#define HF_STDOUT 1       /* Standard output handle */
#define BUF_SIZE 80

BYTE    abBuffer[BUF_SIZE];
HFILE   hf,
        hfNew;
ULONG   ulRead,
        ulWritten,
        ulAction;
APIRET  ulrc;

ulrc = DosOpen("SAMPLE.C",
               &hf,
```

```
                &ulAction,
                0,
                FILE_NORMAL,
                FILE_OPEN,
                OPEN_ACCESS_READONLY |
                OPEN_SHARE_DENYNONE,
                (PEAOP2) NULL);

if (!ulrc) {

    hfNew = 0;      /* Duplicates standard input */

    DosDupHandle(hf,
                 &hfNew);

    DosRead(HF_STDIN,
            abBuffer,
            sizeof(abBuffer),
            &ulRead);

    DosWrite(HF_STDOUT,
             abBuffer,
             ulRead,
             &ulWritten);
}
```

File Names

File names are the identifiers used by the file system to uniquely identify files on a disk. All file systems have specific rules for constructing names of file objects. Different file systems can have different rules for naming file objects.

The OS/2 FAT file system supports the DOS naming conventions. The OS/2 High Performance File System (HPFS) supports a superset of the DOS naming conventions, allowing for long file names and characters illegal under DOS. Although different file systems can have different rules for naming file objects, all OS/2 file systems require that full path names consist of directory and file names separated by backslashes (\).

The OS/2 operating system views path names as ASCII strings and does not restrict file systems to the DOS file name format. Compatibility with existing DOS applications requires that all installable file systems support a superset of the 8.3 file name format used in the FAT file system.

The following topics are related to the information in this chapter:

- File Systems
- File Management
- Extended Attributes
- Device I/O

File-Naming Conventions

File name conventions are the rules used to form file names in a given file system. Although each installable file system (IFS) can have specific rules about how individual components in a directory or file name are formed, all file systems follow the same general conventions for combining components. For example, although the FAT file system requires that file and directory names have the 8.3 file name format, and HPFS supports names of up to 255 characters long, both file systems use the backslash (\) character to separate directory names and the file name when forming a path.

When creating names for directories and files, or when processing names supplied by the user, an application must follow these general rules:

- Process a path as a NULL-terminated string. An application can determine maximum length for a path by using DosQuerySysInfo.
- Use any character in the current code page for a name, but do not use a path separator, a character in the range 0 through 31, or any character explicitly prohibited by the file system.

 The following characters are reserved by the operating system. Do not use them in directory or file names.

```
<   >   :   "   /   \   |
```

Although a name can contain characters in the extended character set (128 - 255), an application must be able to switch code pages if necessary to access the corresponding file.

- Compare names without regard to case. Names such as "ABC", "Abc", and "abc" are considered to be the same.
- Use the backslash (\) or the forward slash (/) to separate components in a path. No other character is accepted as a path separator.
- Use the dot (.) as a directory component in a path to represent the current directory.
- Use two dots (..) as a directory component in a path to represent the parent of the current directory.
- Use a period (.) to separate components in a directory name or file name. Unless explicitly defined by a file system, no restrictions are placed on the number of components in a name.

File Names in the FAT File System

Valid file names in the OS/2 FAT file system have the following form:

```
[drive:][directory\]filename[extension]
```

The *drive* parameter must name an existing drive and can be any letter from A through Z. The drive letter must be followed by a colon (:).

The *directory* parameter specifies the directory that contains the file's directory entry. The directory name must be followed by a backslash (\) to separate it from the file name. If the specified directory is not the current directory, *directory* must include the names of all the directories in the path, separated by backslashes. The root directory is specified by using a backslash at the beginning of the name.

For example, if the directory ABC is in the directory SAMPLE, and SAMPLE is in the root directory, the directory specification is:

```
\SAMPLE\ABC.
```

A directory name can also have an extension, which is any combination of up to three letters, digits, or special characters, preceded by a period (.).

The *filename* and *extension* parameters specify the file.

FAT File-Naming Rules
For file objects managed by the FAT file system, the following rules apply:

- File names are limited to 8 characters before and three characters after a single dot. This is referred to as the 8.3 file name format.

 The 8 characters before the dot are blank-filled. Embedded blanks are significant, trailing blanks and blanks immediately preceding the dot are not significant. Trailing blanks are truncated.

 For example, "FILE.A" is really "FILE .A ". "FILE.A" and "FILE .A " are treated as the same file by the operating system and refer to the same file. Also, "FILE.TXT " and "FILE.TXT" are treated as the same file.

 Blanks elsewhere in the name are significant-"F I L E.TXT" is not the same as "FILE.TXT".
- Names are not case sensitive. This means that "FILE.TXT" and "file.txt" refer to the same file. Lowercase and uppercase characters are folded together for name comparison purposes.
- Names returned by file system functions are in uppercase. This means that if "file.txt" is created, DosFindFirst returns "FILE.TXT".
- Directory and file names can be any combination of up to eight letters, digits, or the following special characters:

  ```
  $   %   '   -   _   @   {   }   ~   `   !   # (   )
  ```

 File extensions can be any combination of up to three letters, digits, or special characters, preceded by a period.
- Invalid characters for directory names, file names, and volume labels are:
 - the range 0 - 1Fh

- and the characters:

```
< > | + = : ; , . " / \ [ ]
```

File Names in the High Performance File System

In HPFS, file names can be up to 255 characters long (one must be a terminating NULL, "\0"). Directory names can also be 255 characters long, but the length of the complete path, including drive, directories, and file name, cannot exceed 260 characters.

Certain characters that are illegal in the FAT file system are legal in HPFS file names:

```
+ = ; , [ ]
```

Also, blank spaces can be used anywhere in an HPFS file name or directory name, but blank spaces and periods at the end of a file name are ignored. Additionally, the period (.) is a valid file name character and can be used as many times as desired. There is no requirement that HPFS file names have extensions; however, many applications still create and use them.

An HPFS file name can be all uppercase, all lowercase, or mixed case. The case is preserved for directory listings but is ignored in file searches and all other system operations. Therefore, in a given directory, there cannot be more than one file with the same name when the only difference is case.

File-Naming Rules for Installable File Systems
For file objects managed by OS/2 installable file systems, the following rules apply:

- Each element of a full path name residing on a disk managed by an installable file system can consist of up to 255 characters. File names can be up to 255 characters long (one of the characters must be a terminating NULL, "\0"). Directory names can also be 255 characters long, but the length of the complete path, including drive, directories, and file name, cannot exceed 260 characters. For example, in the path name "c:\XXX...XXX\YYY", "XXX...XXX" can include up to 255 characters. This is referred to as long file names.
- Names are not case sensitive.
- File name case as specified at create time is preserved. This means that if the file "file.TXT" is created, DosFindFirst returns "file.TXT". File name case may be modified using DosMove.
- Blanks immediately preceding a dot are significant. This means that "FILE.TXT" and "FILE .TXT" refer to different files.
- Trailing blanks are truncated. This means that "FILE.TXT " is the same as "FILE.TXT".
- Blanks elsewhere in the name are significant. This means that "F I L E.TXT" is not the same as "FILE.TXT".
- For compatibility reasons, trailing dots on component names are discarded. For Example, "\FILE.TXT...TEXT...\A..B...\C." becomes "\FILE.TXT...TEXT\A..B\C". This processing includes semaphore, queue, pipe, module, shared memory names, and device names.
- The set of legal characters is expanded to include

  ```
  + = ; , [ ]
  ```

 as well as all characters legal for the FAT file system.
- If an installable file system uses a component separator within a file name, it must be a dot (.). There are no restrictions on the number of components which can be allowed within a file name, for example "My.Programming.Reference.Part.One".

Long File Names

Programs that recognize long file names must indicate this by including the NEWFILES statement in their module definition file. This statement directs the linker to set a bit in the executable file header. It indicates that the module supports long file names. This bit is meaningless in a DOS Session and on versions of the OS/2 operating system prior to Version 1.2. Programs written for OS/2 Version 1.2 (and all later versions) installable file systems should set this bit. Bound programs that have this bit set can see files with long file names in OS/2

mode, but only files with 8.3 file name format in DOS Sessions.

This bit has meaning when attached to program modules, not when attached to DLLs. Whether the program recognizes long file names format is entirely dependent on the value of its NEWFILES bit and the effect of the bit extends into any calls to DLLs. In order to be compatible with all OS/2 file systems, dynamic link libraries must not create internal temporary files or directories that do not comply with 8.3 file naming conventions. In addition, dynamic link libraries cannot return long file names to an application. (The caller might be running on a file system that only supports 8.3 file names and use the returned name to create a file.)

OS/2 applications which do not recognize long file names can run with some restrictions. For these programs, long names (including device names) are filtered according to the following rules:

- Any name not representable in the 8.3 file name format is not returned from DosFindFirst or DosFindNext. This is because the application's buffers are unlikely to be large enough to handle longer names.
- Any long file name passed to the file system functions listed below are rejected in exactly the same way as under previous versions of the OS/2 operating system. It is not acceptable to create and manipulate a name that you cannot find.

 DosOpen
 DosDelete
 DosMove
 DosQueryPathInfo
 DosSetPathInfo
 DosCreateDir
 DosDeleteDir
 DosFindFirst
 DosFindNext
 DosQueryFSAttach
 DosFSAttach
 DosCopy
 DosSearchPath

- Long file names can be passed to DosSetCurrentDir and DosQueryCurrrentDir so that all programs can use all directories.
- Long names used with non-file system functions (for example, DosCreateSem) are not filtered.

For files located on file devices managed by the OS/2 FAT file system, long file names are handled differently in OS/2 mode than in DOS mode. In OS/2 mode, the long file name is considered an error. In DOS mode, the name is truncated and is not an error. The DOS mode treatment of file name formats provides compatibility with the PC-DOS environment for applications originally written for PC-DOS. However, if you are writing a family application to run under both the OS/2 operating system and the PC-DOS environment, your application must allow for this difference in operating environments.

Because long file names can be input to applications through program command lines, dialog boxes, or function calls, applications must provide their users with rules for how to enter file names. File Names in User Input provides some general guidelines in this matter, that are applicable to both long file names and 8.3 file names.

Moving Files with Long Names

The Workplace Shell supports copying files with long file names to media that is managed by a non-installable file system (IFS) and for returning these files to IFS media with the long name intact.

When a file with a long name is copied to media that does not support long file names, the Workplace Shell stores the file's long name in the .LONGNAME extended attribute. When the file is copied back to a disk that does support long file names, the Workplace Shell restores the long name from the extended attribute.

If the new media does not support extended attributes, files that have long names cannot be moved to the media without having their names modified or truncated.

Note: The behavior described above only applies to the Workplace Shell The command processors, CMD.EXE and COMMAND.COM, do not automatically save the long file name; they require the user to enter a new file name that is legal on the new media. The DosCopy command also does not save the long file name automatically; the programmer must provide the target file name to DosCopy and the target file name must be a legal file name for the target media.

If you choose to store and restore the file's long name, you must do it yourself in the manner described above.

Metacharacters in File Names

Metacharacters are characters that can be used to represent placeholders in a file name. The asterisk (*) and the question mark (?) are the two metacharacters.

The asterisk matches one or more characters, including blanks.

The question mark matches exactly one character, unless that character is a period. To match a period, the original name must contain a period. Metacharacters are illegal in all but the last component of a path.

Metacharacters are also referred to as global file name characters, or as wildcard characters.

An application that allows more than one file name on its command line, can accept metacharacters to provide users with a shortcut for entering a long list of names. For instance, metacharacters can be used to reference a set of files with a common base name; to reference all files with an extension of EXE, the user would enter:

```
*.exe
```

Although a name that contains metacharacters is not a complete file name, an application can use functions, such as DosFindFirst and DosEditName, to expand the name (replace the metacharacters) and create one or more valid file names.

Metacharacters have two sets of semantics:

- As *search metacharacters* , which are used to select the files that are returned to the user when the user searches the disk for a file.
- As *edit metacharacters* , which are used to construct a new file name, given a source name and a target name specification.

Both asterisks and question marks, therefore, have two sets of rules, one for searching for file names and one for editing file names.

Search metacharacters are used in commands that search for files or groups of files, like DIR:

```
dir *.exe
```

An application can expand a name with metacharacters to a list of file names by using DosFindFirst and DosFindNext. These functions take a file name template (a name with metacharacters) and return the names of files on the disk that match the pattern in the template.

Edit metacharacters are used in commands that can change the names of files; for example, in a global copy command:

```
copy *.txt  *.old
```

An application can create a new file name from an existing name by using the DosEditName function. This function takes a template (a name with metacharacters) and expands it, using characters from an existing name. An asterisk in the template directs the function to copy all characters in the existing name until it locates a character that matches the character following the asterisk. A question mark directs the function to copy one character, unless that character is a period. The period in the template directs the function to look for and move to the next period in the existing name, skipping any characters between the current position and the period.

Searching for Files Using Metacharacters

An asterisk (*) matches 0 or more characters, any character, including blank. It does not cross NULL or \, which means it only matches a file name, not an entire path.

A question mark (?) matches 1 character, unless what it would match is a period (.) or the terminating NULL, in which case it matches 0 characters. It also does not cross the backslash character (\).

Any character, other than asterisks and question marks, matches itself, including a period.

Searching is case-insensitive. For example, "FILE.TXT" references the same file named "file.txt".

For compatibility reasons, any file name that does not have a dot in it gets an implicit one automatically appended to the end during searching operations. This means that searching for "FILE." would return "FILE".

Some file system functions accept file object name specifications using metacharacters.

Editing File Names Using Metacharacters

Metacharacters in a source name simply match files and behave just like any other search metacharacter.

Metacharacters in a target name are copy-edit commands and work as follows:

- A question mark (?) copies one character unless the character it would copy is a period (.), in which case it copies 0 characters. It also copies 0 characters if it is at the end of the source string.
- An asterisk (*) copies characters from the source name to the target name until it finds a source character that matches the character following it in the target.
- A period (.) in the target name causes the source pointer to match the corresponding "." in the target. They count from the left.

Editing is case-insensitive. If a case conflict between the source and editing string arises, the case in the editing string is used, thus:

```
copy file.txt *E.tmp
```

results in file.txt being copied as filE.tmp.

DosEditName provides applications with the ability to transform a file object name into another name, using an editing string that contains global characters.

Transforming File Names Using Metacharacters

File system functions that an application uses to copy, rename or move file objects do not support the use of global characters. For example, a user can perform a global copy of all files with the extension .EXE by entering the following on the command line:

```
copy *.exe
```

An application, however, cannot perform a similar global copy operation by making a single call to DosCopy or DosMove. These functions operate on a single, specific file object.

DosEditName, however, provides applications with the ability to transform an element of a full path name into another name, using an editing string that contains global characters. For example, for an application to copy all files with an extension of .SRC to files with an extension of .SAM, the application would:

1. Search for all files with the .SRC extension by using DosFindFirst and DosFindNext,
2. Transform the file names by using DosEditName with an editing string of "*.SAM",
3. Copy the files with the new extension with DosCopy.

File Names in User Input

Users often supply file names as part of an application's command line or in response to a prompt from the application. Traditionally, users have been able to supply more than one file name by separating the names with certain characters, such as a blank space. In some file systems, however, traditional separators are valid file name characters. This means additional conventions are required to ensure that an application processes all characters in a name.

When an application processes arguments (including file names) from its command line, the operating system treats the double quotation mark (") and the caret (^) as quotation characters. All characters between the opening and closing double quotation marks are processed as a single argument. The caret is used to quote characters that would otherwise have some special property. The character immediately following the caret is treated as a normal character; any special characteristics that the character has are to be ignored. For example, the greater-than symbol (>) normally causes a program's output to be redirected to a file or device. Typing "^>" causes the ">" to be included in the command line passed to the application. In both cases, the operating system discards the quotation characters and does not treat them as part of the final argument.

When a Presentation Manager*(PM) application processes two or more file names from a dialog box or other prompt, it expects the user to enter each file name on a new line. Therefore, a PM application would use a multiple-line entry field to prompt for multiple file names. This often makes the use of quotation characters unnecessary.

When an application is started, the operating system constructs a command line for the application. If the command line includes file names,

the operating system places a space character between names and marks the end of the list with two NULL characters. Applications that start other applications by using DosExecPgm can also pass arguments by using this convention or by using quotation characters. In practice, most applications receive a command line as a single, NULL-terminated string. Therefore, applications that use DosExecPgm should prepare command lines as a single string, and enclose any file names in quotation marks.

Device Names

Naming conventions for character devices are similar to those for naming files. The OS/2 operating system has reserved certain names for character devices supported by the base device drivers. These device names are listed below:

CLOCK$	Clock
COM1-COM4	First through fourth serial ports
CON	Console keyboard and screen
KBD$	Keyboard
LPT1	First parallel printer
LPT2	Second parallel printer
LPT3	Third parallel printer
MOUSE$	Mouse
NUL	Nonexistent (dummy) device
POINTER$	Pointer draw device (mouse screen support)
PRN	The default printer, usually LPT1
SCREEN$	Screen

These names can be used with DosOpen to open the corresponding devices. Reserved device names take precedence over file names; DosOpen checks for a device name before checking for a file name. Do not use a file name which is the same as a reserved device name; the file will never be opened, because the command will open the device instead.

COM1 through COM4 are reserved device names only when the ASYNC (RS-232C) device driver is loaded. The same is true for POINTER$ and MOUSE$, which are reserved only when a mouse device driver is loaded.

An application can call DosQueryFHState to verify that a file or device has been opened. See Determining and Setting the State of a File or Device Handle for more information on getting the state of a file handle.

File Systems

An application views a file as a logical sequence of data; OS/2 file systems manage the physical locations of that data on the storage device for the application. The file systems also manage file I/O operations and control the format of the stored information.

Applications use the OS/2 file system functions to open, read, write, and close disk files. File system functions also enable an application to use and maintain the disk that contains the files-the volumes, the directories, and the files on the disks of the computer. Applications also use OS/2 file system functions to perform I/O operations to pipes and to peripheral devices connected to the computer, like the printer.

The following topics are related to the information in this chapter:

- File names
- File management
- Extended attributes
- Device I/O

About File Systems

A file system is the combination of software and hardware that supports storing information on a storage device. In the OS/2 operating system, the file system specifies how data is organized on the mass storage devices of the computer, such as hard disks and floppy disks.

Each drive is assigned a unique letter to distinguish it from other drives. A single hard disk can also be divided into two or more *logical drives* . A logical drive represents a portion of the hard disk and, like a physical drive, is assigned a unique letter to distinguish it from other physical and logical drives.

The file system organizes disks into volumes, directories, and files. A volume is the largest file system unit. It represents all the available storage on the logical drive. An optional volume name identifies the volume.

Volumes are subdivided into directories, which contain files and other subdirectories. Each volume has a root directory, which contains file and directory entries. All other subdirectories trace their ancestry back to the root directory. Each directory entry identifies the name, location, and size of a specific file or subdirectory on the disk. A file is one or more bytes of data stored on the disk. Subdirectories provide an additional level of organization and, like the root directory, can contain files and directory entries.

The file system also enables users and applications to access certain non-disk devices as if they were files. An example of such a device would be the printer, which can be accessed through the file system by using the printer's logical name, PRN, as a file name.

Types of File Systems

The OS/2 operating system has two file systems: the *file allocation table (FAT) file system* and the *high performance file system* (HPFS). These file systems define how information is organized on the storage devices.

A user can choose to install either or both file systems. An application must be able to work with any file system. The OS/2 operating system provides a common set of file system functions that are not dependent on a particular file system.

Both of these file systems support:

- Existing logical file and directory structure
- Existing naming conventions
- Multiple logical volumes (partitions)
- Multiple and different storage devices
- Redirection or connection to remote file systems
- Extended attributes
- Metacharacter file-name processing.

Additionally, HPFS supports:

- Long file names
- An extendable application interface.

The High Performance File System is an example of a class of file systems called *installable file systems* . Installable file systems are installed by the user (by changing CONFIG.SYS) and are loaded by the operating system during system initialization.

The OS/2 operating system permits users to have multiple file systems active at the same time; for example a FAT file system for one hard disk and HPFS for another.

FAT File System

The OS/2 FAT file system is based on the DOS FAT file system. This file system, also used in previous releases of the OS/2 operating system and in PC-DOS, controls storage of data files for hard and floppy disks.

The FAT file system is hierarchical, supporting multiple directories on the disk. Each directory can contain one or more files or subdirectories.

The FAT file system uses the 8.3 file name convention. Under this convention, the file name consists of a file name of up to eight characters, a separating period (.), and a file name extension of up to three characters.

Installable File Systems

An installable file system is a file system in which software is installed when the operating system is started. The OS/2 operating system supports installable file systems and permits users to have multiple file systems active at the same time.

Users install a file system by specifying the file system components in the CONFIG.SYS file. The file system software consists of device drivers and dynamic link libraries. The device drivers access storage devices; the dynamic link libraries control the format of information on a device and manage the flow of data to and from the device. The user must use the DEVICE= command to specify the device driver and the IFS= command to specify the dynamic link library.

Installable file system drivers are loaded during system initialization when an IFS= statement is encountered in the CONFIG.SYS file. The operating system loads the device driver and dynamic link library and initializes a specific device for use with a file system.

These file systems can support file and directory structures different from the FAT file system.

An example of an installable file system might be a file system designed specifically for use on a network server. Another example of an installable file system is the High Performance File System (HPFS), which is included with the OS/2 operating system.

High Performance File System

The High Performance File System (HPFS) is an installable file system. It is a hierarchical system and supports multiple directories. In many cases, accessing files under HPFS is faster than accessing similar files under the FAT file system. During installation of the OS/2 operating system, users can install the HPFS on the hard disk they use to start their computer.

Features of HPFS include:

- Caching of directories, data, and file system data structures
- Multi-threaded I/O operations
- Write-behind logic
- Optional write-through
- Strategic allocation of directory structures
- Highly contiguous file allocation
- Enhanced recoverability
- Extended attribute support
- Long file name support
- Starting the OS/2 operating system from an HPFS disk

File names under HPFS can contain 255 characters (one must be the terminating NULL, "\0") and can contain characters that are not valid for the FAT file system-for example, spaces. Each element of a path name residing on an HPFS disk can also have up to 255 characters. The total path including drive, directories, and file name cannot exceed 260 characters (259 with the terminating NULL). For more information on long file name support by installable file systems see Long File Names.

HPFS provides extremely fast access to very large disk volumes. HPFS uses a memory cache divided into blocks of 2KB. Data that is read from and written to the disk is transferred through this cache. Frequently-used data will often be found in the cache, thereby saving the time that a disk-read operation would require. When a user request specifies data that is not present in the cache, HPFS selects the least-recently used discardable block and then fills the block with the requested data.

When a write-data request is received, it usually is not necessary that the data be immediately written to the disk. HPFS will copy such data into the block cache without actually performing the disk-write operation. When the data is in the cache, it is written to disk as a background activity (referred to as lazy writing) which enables the typical user-write operation to occur much faster than in file systems where all write operations are synchronous.

The High Performance File System consists of:

- The High Performance File System driver, HPFS.IFS
- The High Performance File System lazy-write utility, CACHE.EXE
- The High Performance File System lazy-write startup program, STARTLW.DLL
- The High Performance File System utilities, UHPFS.DLL

The user determines the amount of lazy-write support by setting the following parameters on the command line that calls CACHE.EXE:

- MaxAge:

 When the data in a cache block exceeds the specified time the block is queued for writing to the disk. This reduces the amount of data lost due to inopportune system shutdowns.

- DiskIdle and BufferIdle

 When no user I/O request (non-lazy-write) has been made for *DiskIdle* number of milliseconds, all cache blocks (in oldest-first order) that have not been touched for *BufferIdle* number of milliseconds are queued for writing to disk. This enables HPFS to write out user data during times of relative disk inactivity and to reduce the need for rewriting heavily used cached blocks.

STARTLW.DLL contains the code that starts the lazy-write thread.

Local and Remote File Systems

Installable file systems work with a variety of storage devices. A file system on a local device such as a disk drive or virtual drive is called a local file system. A file system on a remote device, such as a disk drive on another computer, is called a remote file system. An application can establish a connection to a local or a remote file system by using DosFSAttach.

For a local file system, the operating system uses a block device driver, which accesses disk hardware, to handle input and output to the device. The operating system automatically connects most (if not all) local file systems when it starts. However, an application can attach and detach additional file systems as needed.

For a remote file system, the operating system uses a device driver that typically accesses a communications or network device. Usually, the actual storage device is located on another computer, and the two computers communicate requests and data through a network connection. An application can associate a remote file system with a drive letter by using DosFSAttach. Once the connection is made, the application can access directories and files on the remote device simply by using the assigned drive letter, treating the remote device as if it were on the same computer.

Recognizing DOS and OS/2 File Objects

The OS/2 FAT file system recognizes file objects created by the DOS FAT file system. This means that applications running under the OS/2 operating system (these include both OS/2 applications and DOS applications running in a DOS Session) can access file objects created by applications running under DOS.

Because the OS/2 FAT file system supports the same directory structure as the DOS FAT file system, applications running under DOS can access files and directories created by the OS/2 FAT file system.

However, the High Performance File System (HPFS) does not support the same directory structure as the DOS FAT file system. Therefore, the DOS FAT file system will not recognize file objects created by HPFS. This means that if you start the computer with DOS, applications running under DOS cannot access files and directories on HPFS disks.

DOS applications running in a DOS Session under the OS/2 operating system *can* recognize files and directories on both FAT and HPFS disks. A request from a DOS Session to read a file on a FAT disk is handled by the OS/2 FAT file system. Similarly, a request from a DOS Session to read a file on an HPFS disk is handled by the OS/2 High Performance File System.

Storage Devices and File Systems

OS/2 file systems store information on mass storage devices. These devices are usually hard disks or floppy diskettes, but can be other media, such as CD-ROM.

Each drive (or device) is assigned a unique letter to distinguish it from other drives. On most personal computers, drive A is the first floppy disk drive, drive B is the second floppy disk drive, drive C is the first hard disk drive, and drive D is the second hard disk drive.

A single hard disk can be divided into two or more partitions, each of which are then viewed as a separate logical drive. A logical drive, like a physical drive, is assigned a unique letter to distinguish it from other physical and logical drives. FDISK is the OS/2 utility used to partition physical storage devices.

A personal computer running the OS/2 operating system can have up to 26 logical disk drives.

Each logical storage device can be managed by a different file system. The file system attached to a storage device manages that device. A user attaches a file system to a storage device by:

- Loading the file system driver during system initialization (by including an IFS= statement in CONFIG.SYS).
- Formatting the storage device by using the format options for the file system.

During installation of the OS/2 operating system, users have the option of formatting hard disks with the FAT file system or with the (HPFS). If the user chooses to use the HPFS, an IFS= statement is added to the CONFIG.SYS file so that HPFS is loaded automatically during each system startup. During formatting, the file system driver is associated with the logical storage device or drive letter of the hard disk.

When an application calls a file system function, the operating system directs the request to:

- The installable file system managing the storage device, or
- The FAT file system, if no installable file system is loaded and attached to the storage device.

The file system used to format the storage media manages that media each time the system is started, as long as the file system is loaded during system start-up. The operating system directs file system requests for a storage media to the file system that formatted the media. If no file system recognizes the format of the media, the OS/2 FAT file system attempts to manage that media. This might occur when the file system used to format the storage media is not loaded during system startup (the IFS= statement was removed from the CONFIG.SYS file after OS/2 installation). If the OS/2 FAT file system cannot recognize the media format (the media might have a different directory structure), the user receives an error when attempting to access the media.

For example, assume a system is configured with diskette drive A and hard disk drives C and D. During OS/2 installation, the user elects to format drive C using HPFS. Drive C is, then, managed by HPFS. Drive D was formatted with the FAT file system, so it is managed by the OS/2 FAT file system, as is diskette drive A (removable media cannot be formatted using HPFS). When an application calls DosOpen to open

a file on drive C, the operating system directs the request to HPFS. When an application calls DosOpen to open a file on drive A, the operating system directs the request to the OS/2 FAT file system.

If HPFS is not loaded during system startup, the FAT file system will receive file system requests made for drive C. However, because HPFS supports a different directory structure than the FAT file system does, the OS/2 FAT file system cannot recognize file objects on the disk. The user will receive an error when attempting to gain access to the disk.

Users can determine which file system was used to format a storage device by using the CHKDSK utility. CHKDSK displays a message indicating which file system manages the specified drive. Because DOS does not use extended attributes, a user must use CHKDSK in an OS/2 session rather than in a Dos Session to examine a FAT partition.

File System Utilities

Utilities for each file system are in a single dynamic link library. The utilities that the operating system calls are based on the file system that recognizes the volume on which the utility is to be run. The dynamic link library for each file system has the following utilities:

FORMAT	Disk formatter
CHKDSK	File system validation and repair
RECOVER	File recovery
SYS	System installation

OS/2 Boot Manager

The OS/2 *Boot Manager* enables different operating systems to co-reside on the same computer. The user selects the operating system to boot when the computer is turned on. For example, DOS, AIX, and the OS/2 operating system can co-reside on the same machine. There can also be a previous version of the OS/2 operating system on the machine co-existing with the current version of the operating system.

Each operating system has its own partition and each partition is managed by the appropriate file system for the operating system that owns it. A DOS partition has a FAT file system. An OS/2 partition can have either a FAT file system or HPFS. An AIX partition will use the AIX file system to manage its partition.

Note: FAT partitions that follow HPFS partitions on the same physical disk cannot be accessed when using DOS because DOS stops at the first partition it does not recognize.

Using File Systems

In order to take advantage of the capabilities of OS/2 file systems, application developers must be able to manage the file systems.

Most of the file system functions work with either the FAT file system or the High Performance File System (HPFS).

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Attaching and Detaching File Systems

A file system driver that uses a block device driver for I/O operations to a local or remote (virtual disk) device is called a local file system. A file system driver that accesses a remote system without a block device driver is called a remote file system.

An application, typically a network application, can call DosFSAttach to:

- Attach a drive to a remote file system
- Detach a drive from a remote file system
- Attach a pseudocharacter device name to a local or remote file system
- Detach a pseudocharacter device name from a local or remote file system.

DosFSAttach establishes or breaks the connection between a drive or device and a file system. If an attachment is successful, all requests to that drive or name are routed to the specified file system. If a detachment is successful, the operating system will no longer recognize the drive or name in a file system call.

DosFSAttach does not support:

- Redirection of drive letters representing local drives
- Attachment to drives or devices that are not in the system's name space. (DosFSCtl can be used to attach to drives or devices not in the system's name space.)

 A name space is a set of names that are known to the file system. For example, CON (console) and PRN (printer) are always in the OS/2 file system's name space.

The following code fragment attaches a drive to a remote file system driver (FSD). Assume that the FSD does not require any user-supplied data arguments.

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

UCHAR   ucDeviceName[8]; /* Device name or drive letter string */
UCHAR   ucFSDName[40];   /* FSD name                           */
PVOID   pDataBuffer;     /* Attach argument data               */
ULONG   ulDataBufferLen; /* Buffer length                      */
ULONG   ulOpFlag;        /* Attach or detach                   */
APIRET  ulrc;            /* Return code                        */

strcpy(ucDeviceName,"Y:");
                    /* Drive letter with which to attach the */
                    /* file system driver                    */

strcpy(ucFSDName,"\\lan03\\src");

pDataBuffer = 0;      /* Assume that no user-supplied data     */
                      /* arguments are required                */

ulDataBufferLen = 0;  /* No data buffer supplied               */

ulOpFlag = 0;         /* Indicate Attach request               */

ulrc = DosFSAttach(ucDeviceName,
                   ucFSDName,
                   pDataBuffer,
                   ulDataBufferLen,
                   ulOpFlag);

if (ulrc != 0) {
    printf("DosFSAttach error: return code = %ld", ulrc);
    return;
}
```

Obtaining Information about an Attached File System

To obtain information about block devices, and all character and pseudocharacter devices, including the type of device and the name of the file system driver the device is attached to, use DosQueryFSAttach.

The information can be used to determine if the operating system recognizes that a particular file system is attached to a storage device. This is important to an application that must guarantee such a state. An application of this type must handle the situation where the file system driver that formatted a certain disk was not loaded during system startup. (The user might have omitted the IFS= statement in the CONFIG.SYS. file). In such a situation, the data on the disk could be destroyed because the wrong file system was attached to the disk by default.

The following code fragment returns information about an attached file system.

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

UCHAR       ucDeviceName[8];  /* Device name or drive letter     */
                             /* string                         */
ULONG       ulOrdinal;        /* Ordinal of entry in name list  */
ULONG       ulFSAInfoLevel;   /* Type of attached FSD data      */
                             /* required                       */
FSQBUFFER2  fsqDataBuffer;    /* Returned data buffer           */
ULONG       ulDataBufferLen;  /* Buffer length                  */
APIRET      ulrc;             /* Return code                    */

strcpy(ucDeviceName,
       "Y:");  /* Logical drive of the attached file system */

ulFSAInfoLevel = 1;

ulDataBufferLen = sizeof(FSQBUFFER2);    /* Data buffer length   */

ulrc = DosQueryFSAttach(ucDeviceName,
                        ulOrdinal,
                        ulFSAInfoLevel,
                        &fsqDataBuffer,
                        &ulDataBufferLen);

if (ulrc != 0) {
    printf("DosQueryFSAttach error: return code = %ld", ulrc);
    return;
}
```

In this example, information was requested about the drive whose name was specified within the *DeviceName* variable. After the DosQueryFSAttach call, the *DataBuffer* structure contained a set of information describing the specified attached file system, and the *DataBufferLen* variable contained the size of information within the structure.

Obtaining Information about a File System

An application can retrieve information about the file system on a given drive by using DosQueryFSInfo. The file system information includes information on the amount of free storage space on the disk. The storage space is given in number of allocation units (clusters) on the disk. Each cluster has an associated number of sectors; each sector contains a given number of bytes. A typical disk has 512 bytes for each sector and 4 sectors for each cluster. DosSetFSInfo enables an application to change the volume identifier for the disk in the given drive.

The following code fragment obtains information about the file system that is associated with a particular logical drive.

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>
#include <stdio.h>

ULONG   ulDriveNumber;     /* Drive number                 */
ULONG   ulFSInfoLevel;     /* File system data required    */
UCHAR   ucFSInfoBuf[40];   /* File system info buffer      */
ULONG   ulFSInfoBufSize;   /* File system info buffer size */
APIRET  ulrc;              /* Return code                  */

ulDriveNumber = 3;              /* Specify drive C                      */

ulFSInfoLevel = FSIL_ALLOC;     /* Indicate that file system allocation */
                                /* information is requested             */

ulFSInfoBufSize = 40;           /* Size of return data buffer           */

ulrc = DosQueryFSInfo(ulDriveNumber,
                      ulFSInfoLevel,
                      ucFSInfoBuf,
                      ulFSInfoBufSize);

if (ulrc != 0) {
    printf("DosQueryFSInfo error: return code = %ld", ulrc);
```

```
    return;
}
```

In this example, the data buffer *FSInfoBuf* is used to receive information about space allocation within the specified file system.

Obtaining Information about a File

An application can retrieve and set information about a specific file by using DosQueryFSInfo and DosSetFileInfo. File information consists of the dates and times that the file was created, last accessed, and last written to (only the time and date the file was last written to are given for FAT partitions); the size (in bytes) of the file; the number of sectors (or clusters) the file occupies; and the file attributes.

The following code fragment obtains file information for a specified file. The example obtains the Level 1 information set for the file. The Level 1 information set for a file includes the dates and times of creation, last access, and last writing. It also includes information about the size of the file and the file's standard attributes. Assume that the handle of the desired file has been placed into *FileHandle* already.

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>
#include <stdio.h>

HFILE        hfFileHandle;      /* File handle                 */
ULONG        ulFileInfoLevel;   /* Level of file info required */
FILESTATUS3  fsFileInfoBuf;     /* File info buffer            */
ULONG        ulFileInfoBufSize; /* File data buffer size       */
APIRET       ulrc;              /* Return code                 */

ulFileInfoLevel = 1;  /* Indicate that Level 1 information is desired */

fsFileInfoBufSize = sizeof(FILESTATUS3);
                      /* Size of the buffer that will                */
                      /* receive the Level 1 information             */

ulrc = DosQueryFileInfo(hfFileHandle,
                        ulFileInfoLevel,
                        &fsFileInfoBuf,
                        ulFileInfoBufSize);

if (ulrc != 0) {
    printf("DosQueryFileInfo error: return code = %ld", ulrc);
    return;
}
```

In this example, Level 1 file information is placed into the *FileInfoBuf* buffer.

Communicating with a File System

An extended standard interface between an application and a file system driver is provided by DosFSCtl. This function is similar to DosDevIOCtl, which provides a standard interface between an application and a device driver. An application sends a request to the file system driver by specifying a particular function code. Data is exchanged through data areas and parameter lists.

DosFSCtl can be used to establish open connections to file system drivers that are not attached to a name in the operating system's name space. (A name space is a set of names that are known to the file system. For example, CON and PRN are always in the OS/2 file system's name space.)

The following code fragment demonstrates how a process can communicate with a file system driver (FSD). Assume that the calling process has placed an appropriate file handle into *FileHandle* . Assume that the specified file system recognizes a function code of hex 8100, and that the function code accepts an ASCII string as input, requires no specific command parameter list, and returns a string of ASCII characters to the caller.

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
UCHAR   ucDataArea[100];    /* Data area                          */
ULONG   ulDataLengthMax;    /* Max. length of Data area           */
ULONG   ulDataLengthInOut;  /* Data area length, in and out       */
PVOID   pParmList;          /* Parameter list                     */
ULONG   ulParmLengthMax;    /* Max. length of Parameter list      */
ULONG   ulParmLengthInOut;  /* Parameter list length, in and out  */
ULONG   ulFunctionCode;     /* Function code                      */
PSZ     pszRouteName;       /* Path or FSD name                   */
HFILE   hfFileHandle;       /* File handle                        */
ULONG   ulRouteMethod;      /* Method for routing                 */
APIRET  ulrc;               /* Return code                        */

ulFunctionCode = 0x8100;    /* Indicate the function to request   */
                            /* of the file system                 */

strcpy(ucDataArea,
       "PARM1: 98"); /* ASCII string to pass to file system */

ulDataLengthMax = 100;      /* Tell the file system the maximum   */
                            /* amount of data it can return       */

ulDataLengthInOut = strlen(ucDataArea);
                          /* On input, this is the number of    */
                          /* bytes sent to the file system      */

pParmList = 0;              /* In this example, assume that no    */
ulParmLengthMax = 0;        /* specific command parameter list    */
ulParmLengthInOut = 0;      /* is required by the file system     */
                          /* for this function code             */

ulRouteMethod = 1;          /* Indicate that the file handle      */
pszRouteName = 0;           /* directs routing (this implies      */
                            /* that the RouteName variable is     */
                            /* unused in this example)            */

ulrc = DosFSCtl(ucDataArea,
                ulDataLengthMax,
                &ulDataLengthInOut,
                pParmList,
                ulParmLengthMax,
                &ulParmLengthInOut,
                ulFunctionCode,
                pszRouteName,
                hfFileHandle,
                ulRouteMethod);

if (ulrc != 0) {
    printf("DosFSCtl error: return code = %ld", ulrc);
    return;
}
```

In this example, the the *DataArea* buffer is used to store the ASCII string sent by the file system in response to the function request, and the *DataLengthInOut* variable is used to store the number of bytes placed in the buffer by the file system.

Preparing File Systems for System Shutdown

At any time during normal system operation, data destined for a disk might be in a cache. If this information is not written to disk before the system powered-off, the disk can become corrupted. To prevent this, applications call DosShutdown to ensure that the operating system writes the data in the cache to the disk and prevents any further data from being cached. The user can then safely power-off the system.

Note: This call prepares all file systems and device drivers for system shutdown. Therefore, it must be called only when system shutdown is about to occur. The user and applications will no longer have access to their storage devices.

The following code fragment locks out changes to all file systems, and writes system buffers to the disk in preparation for turning off power to the system.

```
#define INCL_DOSFILEMGR   /* File System values */
#include <os2.h>
#include <stdio.h>
```

```
 ULONG   ulReserved;   /* Reserved, must be zero        */
 APIRET  ulrc;         /* Return code                   */

 ulReserved = 0;       /* Reserved, must be set to zero */

 ulrc = DosShutdown(ulReserved);

 if (ulrc != 0) {
     printf("DosShutdown error: return code = %ld", ulrc);
     return;
 }
```

Writing Cache Buffers

DosResetBuffer is used to write to disk (flush) the file system's cache buffers for a specific file handle. When called with a value of hex FFFF for the file handle, DosResetBuffer writes all files belonging to the requesting process to disk (this usage should be administered with care, so the user is not burdened with insertion and removal of a large number of removable media volumes).

When DosResetBuffer is called for single file handle, the directory entry for the file is updated as if the file had been closed. However, the file remains open.

DosResetBuffer can also be called with the name of a named pipe. The process that calls DosResetBuffer is blocked at one end of the pipe until all data it has written has been successfully read by the process at the other end of the pipe. This enables communicating processes to synchronize their dialogs.

The following code fragment opens a file, writes some data to the file's buffer, then writes the file's system buffer to the disk.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include <os2.h>
#include <stdio.h>               /* Needed for printf   */
#include <string.h>              /* Needed for strcpy   */

int main( USHORT argc, PCHAR argv[] ) {

   HFILE  hfFileHandle   = 0L;       /* Handle for file being manipulated     */
   ULONG  ulAction       = 0;        /* Action taken (returned by DosOpen)    */
   ULONG  ulWrote        = 0;        /* Number of bytes written by DosWrite   */
   APIRET ulrc           = NO_ERROR;            /* Return code                */
   UCHAR  uchFileName[20]  = "MYDATA.DAT",      /* Name of file               */
          uchFileData[100] = " ";               /* Data to write to file      */

   /* Open the file MYDATA.DAT.  If the file already exists, replace it.
      If the file doesn't exist, create it.                          */

   ulrc = DosOpen(uchFileName,                    /* Path and file name         */
                  &hfFileHandle,                  /* File handle                */
                  &ulAction,                      /* Action taken (returned)    */
                  100L,                           /* File primary allocation    */
                  FILE_ARCHIVED |
                  FILE_NORMAL,                    /* File attributes            */
                  OPEN_ACTION_CREATE_IF_NEW |
                  OPEN_ACTION_REPLACE_IF_EXISTS,  /* Open function type         */
                  OPEN_SHARE_DENYREADWRITE |
                  OPEN_ACCESS_READWRITE,          /* Open mode of the file      */
                  0L);                            /* No extended attributes     */

   if (ulrc != NO_ERROR) {
      printf("DosOpen error: return code = %u\n", ulrc);
      return 1;
   }


   strcpy (uchFileData,
           "Data....");     /* This data will be written to file */

   ulrc = DosWrite (hfFileHandle,               /* File handle                */
                    (PVOID) uchFileData,        /* String to be written       */
                    sizeof (uchFileData),       /* Size of string to be written */
                    &ulWrote);                  /* Bytes written (returned)   */
```

```
   if (ulrc != NO_ERROR) {
      printf("DosWrite error: return code = %u\n", ulrc);
      DosClose(hfFileHandle);         /* close the file */
      return 1;
   }

   ulrc = DosResetBuffer(hfFileHandle);

   if (ulrc != NO_ERROR) {
      printf("DosResetBuffer error: return code = %u\n", ulrc);
      DosClose(hfFileHandle);         /* close the file */
      return 1;
   }
   DosClose(hfFileHandle);            /* close the file */
   return NO_ERROR;
}
```

Generic IOCtl Commands

OS/2 device drivers are used to access the I/O hardware. The *IOCtl* functions provide a method for an application, or subsystem, to send device-specific control commands to a physical device driver. These IOCtls are subfunctions that are issued through the DosDevIOCtl API function request. The DosDevIOCtl function request can be used only by OS/2 applications; the INT 21h IOCtl request can be used only by DOS applications.

The category and function fields are as follows. Each code is contained in a byte:

Category	Code
0xxx xxxx	OS/2-defined
1xxx xxxx	User-defined
_xxx xxxx	Code.

Function	Code
0xxx xxxx	Return error, if unsupported
1xxx xxxx	Ignore, if unsupported
x0xx xxxx	Intercepted by the OS/2 operating system
x1xx xxxx	Passed to driver
xx0x xxxx	Sends data and commands to device
xx1x xxxx	Queries data and information from device
___x xxxx	Subfunction.

Notice that the *send/query* data bit is intended only to standardize the function set; it plays no critical role. Some functions can contain both command and query elements. Such commands are defined as *sends data* .

Generic IOCtl Function Table

The list of categories and functions for the generic IOCtl requests are as follows:

```
 Category   Function   Description

     01h               Serial Device Control

                14h    Reserved

                34h    Reserved

                41h    Set Bit Rate

                42h    Set Line Characteristics (stop, parity,
                       data bits)

                43h    Extended Set Bit Rate

                44h    Transmit Byte Immediate

                45h    Set Break OFF
```

	46h	Set Modem Control Signals
	47h	Behave As If XOFF Received (stop transmit)
	48h	Behave As If XON Received (start transmit)
	49h	Reserved
	4Bh	Set Break ON
	53h	Set Device Control Block (DCB) Parameters
	54h	Set Enhanced Mode Parameters
	61h	Query Current Bit Rate
	62h	Query Line Characteristics
	63h	Extended Query Bit Rate
	64h	Query COM Status
	65h	Query Transmit Data Status
	66h	Query Modem Control Output Signals
	67h	Query Current Modem Input Signals
	68h	Query Number of Characters in Receive Queue
	69h	Query Number of Characters in Transmit Queue
	6Dh	Query COM Error
	72h	Query COM Event Information
	73h	Query Device Control Block (DCB) Parameters
	74h	Query Enhanced Mode Parameters
02h		Reserved
03h		Video Control
	70h	Allocate an LDT Selector
	71h	Deallocate an LDT Selector
	72h	Query Pointer Draw Address
	73h	Initialize Call Vector Table
	74h	ABIOS Pass-Through
	75h	Allocate an LDT Selector with Offset
	76h	Allocate an LDT Selector with Background Validation Options
	7Eh	Allocate Video Buffer
	7Fh	Get Address to ROM Font
04h		Keyboard Control
	50h	Set Code Page
	51h	Set Input Mode (Default ASCII)
	52h	Set Interim Character Flags
	53h	Set Shift State
	54h	Set Typematic Rate and Delay
	55h	Reserved

	56h	Set Session Manager Hot Key
	57h	Set KCB
	58h	Set Code Page Number
	59h	Set Read/Peek Notification
	5Ah	Alter Keyboard LEDs
	5Bh	Reserved
	5Ch	Set NLS and Custom Code Page
	5Dh	Create New Logical Keyboard
	5Eh	Destroy Logical Keyboard
	71h	Query Input Mode
	72h	Query Interim Character Flags
	73h	Query Shift State
	74h	Read Character Data Records
	75h	Peek Character Data Record
	76h	Query Session Manager Hot Key
	77h	Query Keyboard Type
	78h	Query Code Page Number
	79h	Translate Scan Code to ASCII
	7Ah	Query Keyboard Hardware ID
	7Bh	Query Keyboard Code Page Support Information
05h		Parallel Port Control
	42h	Set Frame Control (CPL, LPI)
	43h	Reserved
	44h	Set Infinite Retry
	45h	Reserved
	46h	Initialize Parallel Port
	47h	Reserved
	48h	Activate Font
	49h	Reserved
	4Bh	Reserved
	4Ch	Reserved
	4Dh	Set Print-Job Title
	4Eh	Set Parallel Port Write Time-Out Value
	4Fh	Reserved
	50h	Reserved
	51h	Reserved
	62h	Query Frame Control
	63h	Reserved
	64h	Query Infinite Retry
	65h	Reserved
	66h	Query Parallel Port Status

	67h	Reserved
	68h	Reserved
	69h	Query Active Font
	6Ah	Verify Font
	6Bh	Reserved
	6Ch	Reserved
	6Dh	Reserved
	6Eh	Query Parallel Port Write Time-Out Value
	6Fh	Reserved
	70h	Reserved
	71h	Reserved
06h		Light Pen Control
07h		Mouse Control
	50h	Reserved
	51h	Notification of Display Mode Change
	52h	Reserved
	53h	Reassign Current Mouse Scaling Factors
	54h	Assign New Mouse Event Mask
	55h	Reassign Mouse Threshold Values
	56h	Set Pointer Shape
	57h	Unmark Collision Area
	58h	Mark Collision Area
	59h	Specify/Replace Pointer Screen Position
	5Ah	Set OS/2-Mode Pointer Draw Device Driver Address
	5Bh	Reserved
	5Ch	Set Current Physical Mouse Device Driver Status Flags
	5Dh	Notification of Mode Switch Completion
	60h	Query Number of Mouse Buttons Supported
	61h	Query Mouse Device Motion Sensitivity
	62h	Query Current Physical Mouse Device Driver Status Flags
	63h	Read Mouse Event Queue
	64h	Query Current Event Queue Status
	65h	Query Current Mouse Event Mask
	66h	Query Current Mouse Scaling Factors
	67h	Query Current Pointer Screen Position
	68h	Query Current Pointer Shape
	69h	Query Mouse Threshold Values
	6Ah	Query Physical Mouse Device Driver Level/Version
	6Bh	Query Pointing Device ID

08h		Logical Disk Control
	00h	Lock Drive
	01h	Unlock Drive
	02h	Redetermine Media (end format)
	03h	Set Logical Map
	04h	Begin Format
	20h	Block Removable
	21h	Query Logical Map
	22h	Reserved
	40h	Removable Media Control
	43h	Set Device Parameters
	44h	Write Logical Track
	45h	Format and Verify Track
	5Dh	Diskette Control
	5Eh	Reserved
	5Fh	Reserved
	60h	Query Media Sense
	63h	Query Device Parameters
	64h	Read Logical Track
	65h	Verify Logical Track
	66h	Status
09h		Physical Disk Control
	00h	Lock Physical Drive
	01h	Unlock Physical Drive
	44h	Write Physical Track
	63h	Query Physical Device Parameters
	64h	Read Physical Track
	65h	Verify Physical Track
0Ah		Character Device Monitor Control
	40h	Register Monitor
0Bh		General Device Control
	01h	Flush Input Buffer
	02h	Flush Output Buffer
	41h	System Notifications for Physical Device Drivers
	60h	Query Monitor Support
0Ch		Advanced Power Management
	40h	Send Power Event
	41h	Set Power Event Res
	42h	Reserved
	60h	Query Power Status
	61h	Query Power Event

	62h	Query PowerInfo
0Dh-7Fh		Reserved Category Codes
80h		Screen Control
	00h	Get Current Video Memory Bank
	01h	Set Current Video Memory Bank
	02-07Fh	Reserved
	08h	Return Adapter Video Configuration
	09h	Return Manufacturer-Specific Adapter Data
	0Ah	Update Adapter Video Information
	0Bh	Return Linear Address Mapped to Physical Address
	0Ch-07Fh	Reserved
80h		OEMHLP Controls
	00h	Query OEM Adaptation Information
	01h	Query Machine Information
	02h	Query Display Combination Code
	03h	Return Video Fonts
	04h	Read EISA Configuration Information-Subfunction 00
	04h	Read EISA Function Information-Subfunction 01
	05h	Query ROM BIOS Information
	06h	Query Miscellaneous Video Information
	07h	Query Video Adapter
	08h	Query SVGA Information
	09h	Query Memory Information
	0Ah	Query, Display Mode, Query, and Set (DMQS) Information
	0Bh	Access and Query PCI BIOS Information
80h		Adapter Presence Check (TESTCFG.SYS)
	40h	Get Copy of BIOS/Adapter
	41h	Issue an IN I/O Instruction
	42h	Issue an OUT I/O Instruction
	60h	Get Bus Architecture
	61h	Return all POS IDs
	62h	Return all EISA IDs
80h		Resource Manager Commands
	01h	Get Resource Manager Node Data
	02h	Enumerate Resource Manager Nodes
80h		CD-ROM Drive and Disk Control
	40h	Reset Drive
	44h	Eject Disk
	46h	Lock/Unlock Door

	50h	Seek
	60h	Device Status
	61h	Identify CD-ROM Driver
	63h	Return Sector Size
	70h	Location of Drive Head
	72h	Read Long
	78h	Return Volume Size
	79h	Get UPC
80h		High-Resolution Timer Driver
	00h	Query Version
	01h	Get Resolution
	02h	Set Resolution
	03h	Get Pointer to Clock Counter
	04h	Free Pointer to Clock Counter
	05h	Block Until Time Elapses
81h		CD-ROM Audio Control
	40h	Audio Channel Control
	50h	Play Audio
	51h	Stop Audio
	52h	Resume Audio
	60h	Return Audio-Channel Information
	61h	Return Audio-Disk Information
	62h	Return Audio-Track Information
	63h	Return Audio-Subchannel Q Information
	65h	Return Audio-Status Information
81h		Touch-Device-Dependent Driver Control
	50h	Reserved
	51h	Reserved
	52h	Set Calibration Constants
	53h	Read Data
	54h	Set Data Mode
	55h	Set Click-Lock Parameters
	56h	Set Touch Thresholds
	57h	Set Emulation XY Offset
	58h	Set Data Report Rate
	59h	Set Low-Pass Filter
	5Ah	Write Memory Location
	5Bh	Reserved
	5Ch	Reserved
	5Dh	Reserved
	5Eh	Reserved

	5Fh	Reserved
	60h	Get Calibration Constants
	61h	Get Data Mode
	62h	Get Click Lock Parameters
	63h	Get Touch Thresholds
	64h	Get Emulation XY Offset
	65h	Get Data Report Rate
	66h	Get Low Pass Filter
	67h	Read Memory Location
81h		Touch-Device-Independent Driver Control
	50h	Set Coordinate System
	51h	Reserved
	52h	Set Selection Mechanism
	53h	Set Event Mask
	54h	Set Queue Size
	55h	Set Emulation State
	60h	Set Coordinate System
	61h	Reserved
	62h	Get Selection Mechanism
	63h	Get Event Mask
	64h	Get Queue Size
	65h	Get Emulation State
	66h	Get Read Event Queue

Category 01h ASYNC (RS232-C) Control IOCtl Commands

Whenever an IOCtl command calls for a NULL pointer, it is the responsibility of the application to set one up for the appropriate Parameter or Data Packet pointer before calling the physical device driver. IOCtls can be interpreted differently by future releases if the pointer is not a NULL pointer. If a NULL pointer is called for and it is not received by the device driver, it is considered an invalid Parameter or Data Packet value.

The physical device driver services each communications port (COM1, COM2, and so forth) independently. IOCtls issued to the physical device driver for a given port have no effect on any other communications ports that the physical device driver is servicing. The application cannot assume a given timing relationship between when the IOCtls are executed and when data is received or transmitted by the ASYNC hardware. Data Carrier Detect (DCD) is the same signal as Receiver Line Signal Detect (RLSD).

The following is a summary of the Category 01h IOCtl Commands:

Function	Description
14h	Reserved
34h	Reserved
41h	Set Bit Rate
42h	Set Line Characteristics (stop, parity, data bits)

```
43h     Extended Set Bit Rate

44h     Transmit Byte Immediate

45h     Set Break OFF

46h     Set Modem Control Signals

47h     Behave as if XOFF Received (stop transmit)

48h     Behave as if XON Received (start transmit)

49h     Reserved

4Bh     Set Break ON

53h     Set Device Control Block (DCB) Parameters

54h     Set Enhanced Mode Parameters

61h     Query Current Bit Rate

62h     Query Line Characteristics

63h     Extended Query Bit Rate

64h     Query COM Status

65h     Query Transmit Data Status

66h     Query Modem Control Output Signals

67h     Query Current Modem Input Signals

68h     Query Number of Characters in Receive Queue

69h     Query Number of Characters in Transmit Queue

6Dh     Query COM Error

72h     Query COM Event Information

73h     Query Device Control Block (DCB) Parameters

74h     Query Enhanced Mode Parameters
```

Asynchronous (RS232-C) Communications Physical Device Driver

The Asynchronous Communications (ASYNC) device driver enables OS/2 applications to utilize the Serial Communications (RS232-C) device hardware. The physical device driver allows an application program in OS/2 session to support duplex communications while the device driver:

- Services the RS232-C port in an interrupt-driven manner
- Provides transmit and receive queues
- Provides different automatic control modes for the modem control signals
- Provides logical data stream flow control (XON/XOFF) for transmit and receive operations

The user will normally want to use the physical ASYNC device driver either in conjunction with the spooler (for serial printers only) or with an application program that uses the RS232-enabling capabilities of the physical ASYNC device driver coupled with a serial device attached to the system.

The user installs the physical ASYNC device driver by adding a DEVICE= statement to the CONFIG.SYS file.

Hardware Support

The RS232-C ASYNC communications device driver supports any personal computer system based on an 80386SX (or higher)

microprocessor.

IBM PS/2 Micro Channel Adapter Support

The physical device driver supports a maximum of four ASYNC ports on a maximum of two different interrupt levels. The interrupt levels must have ABIOS support, with one unit per Logical ID (LID) for the ASYNC Device ID. The only ASYNC devices supported on IBM PS/2 and the Extended Industry Standard Architecture (EISA) machines are COM1, COM2, COM3, and COM4. These devices correspond to the first four LIDs in the ABIOS common data area that have the architected ASYNC Device ID. These devices also correspond to the first four ASYNC addresses in the ROM BIOS 40: data area.

If a device has capabilities other than ASYNC that cannot be utilized independently of the ASYNC capabilities (for example, as in the Advanced BIOS separate LID architecture), and if Advanced BIOS assigns the device the ASYNC Device ID, then that device can be used only for ASYNC in that power-on session.

If the device is assigned the ASYNC Device ID, and it has additional capabilities beyond supporting the RS232-C port (for example, a built-in modem), the physical device driver does not recognize those additional capabilities (and potential limitations). Also, the physical device driver does not inform any application program of those additional capabilities or limitations. In addition, it does not limit the control of the RS232-C interface or the device to only those modes that are acceptable to the extended hardware capabilities of that RS232-C port.

If the device is not assigned the ASYNC Device ID, it is not supported by this physical device driver. If an ASYNC device is not supported by the OS/2 operating system, but is recognized by Advanced BIOS as an ASYNC Device ID, the physical device driver can recognize and try to use that unsupported device, if it is COM1, COM2, COM3, or COM4.

AT Bus Adapter Support

The physical device driver for the IBM AT bus machines by default, supports two ASYNC ports, COM1 and COM2, each on separate levels. ASYNC ports with the following base I/O addresses are recognized by the physical device driver:

- 3F8H (must generate a level 4 interrupt)
- 2F8H (must generate a level 3 interrupt)

COM3 and COM4 are supported by the command line parameters for COM.SYS. COM3 and COM4 are supported through parameters on the DEVICE= statement in the CONFIG.SYS file. These parameters specify the desired base I/O address and interrupt levels. The physical ASYNC device driver for the AT- bus machine interfaces directly to the hardware and supports:

- IBM AT-bus serial/parallel adapter based on the NS 16450 Universal Asynchronous Receiver Transmitter (UART) device
- Other compatible adapters based on the UART architecture (NS 16550, NS 16550A)

Attachment Support

The ASYNC physical device driver does not provide any support for devices attached to the RS232-C port. The physical device driver provides enabling support for the RS232-C interface itself. Application programs, subsystems, and systems programs provide the support needed to use devices attached to the RS232-C port.

The ability to support a device can be determined by understanding the level of RS232-C interface enabling support the physical device driver provides, along with the characteristics of the attachment hardware in question and the required functions to be supported.

The OS/2 operating system provides a mechanism where one or more additional drivers can be installed to support specific COM ports. This feature might be required for the following reasons:

- To allow an application program to support a special device not adequately supportable with this ASYNC device driver
- To allow additional COM ports (besides COM1-4 on IBM PS/2) to be supported
- To enhance the level of device driver function for a given COM port (This might be required for certain subsystem support.)

RS232-C Interface

The ASYNC interface consists of separate read and transmit lines. There are two separate modem control signals whose output values can be controlled by the physical device driver:

- Data Terminal Ready (DTR)
- Request To Send (RTS)

There are four separate modem control signals whose input values are available to the physical device driver:

- Data Set Ready (DSR)
- Clear To Send (CTS)
- Data Carrier Detect (DCD), also known as Receive Line Signal Detect (RLSD)
- Ring Indicator (RI)

The receive and transmit data lines have the following hardware characteristics:

- Logical 1 (Marking). More negative than -3 Volts. This state could mean no data.
- Logical 0 (Spacing). More positive than +3 Volts. This state could mean break condition.

The modem control signal lines have the following hardware characteristics:

- Function ON, when more positive than +3 Volts.
- Function OFF, when more negative than -3 Volts.

Hardware Support for Extended Hardware Buffering

This capability is a feature of the asynchronous communications port's serial controller device. Serial controllers that support this capability, such as the NS-16550A UART, are present in many IBM PS/2 systems and on a variety of IBM and non-IBM asynchronous communications adapters.

INS 8250, INS 8250-B Considerations

The following hardware defects cannot be compensated for in the physical device driver and can cause indeterminate function when used with the OS/2 physical ASYNC device driver:

Line Control Configurations
: These devices are known to transmit bad data when configured for 5 data bits and 1.5 stop bits.

Receive Character Overrun Errors
: These devices are known to occasionally drop received characters without posting the RECEIVE_OVERRUN error flag. Undetected data loss can result from this hardware deficiency. Application error-correction routines can be implemented to ensure accurate data transmission when these devices are being used.

Spurious Characters at Power-On
: These devices can transmit a single random character at *power-on* . The connected device must not be expecting valid data to be received until after the physical device driver initialization routine has been run.

Supported Bit Rates on 16450 and Compatibles

The NS 16450 and other compatible UART devices (including the 8250- and 16550-Series UARTs) incorporate a Programmable Baud Generator feature that is driven as a function of the following constants:

```
CLOCK     = 1843200    ; crystal frequency
CLOCK/16  = 115200     ; after divider
```

Given these constants, the algorithm for determining which rates are supported is explained in the following examples:

- If 900 bps is specified, the bit rate is exactly 900 because it divides evenly into 115200 (115200/900 = 128). Bit rate, returned by IOCtl ASYNC_GETBAUDRATE, is 900.
- If 901 bps is specified, the bit rate does not change, and the IOCtl fails with an invalid parameter error because it cannot be supported within .01% (115200/901 = 128, 115200/128 = 900, which gives a .1111% error).
- If 907 bps is specified, the bit rate is 907.0866 because it can be supported within .01% (115200/907 = 127, 115200/127 = 907.0866, which gives a .0095% error). Bit rate, returned by ASYNC_GETBAUDRATE, is 907.
- If 110 bps is specified, the bit rate is 110.0287, even though the error is over .01% (115200/110 = 1047, 115200/1047 = 110.0287, which gives a .0260% error). Bit rate, returned by ASYNC_GETBAUDRATE, is 110.

Note: Where division is performed and the quotient is not a whole integer, an integer result is obtained by rounding off the fractional part of the quotient.

ASYNC (RS232-C) Device Driver Features

The device driver supports the ASYNC interface in an interrupt-driven manner. This allows the multitasking capabilities of the OS/2 operating system to be supported while ASYNC data reception and transmission are taking place.

Warning: With any supported hardware, the physical device driver cannot absolutely guarantee accurate function, as there is a dependency on the hardware being driven. It is known, for example, that INS 8250 and INS 8250-B UART devices exhibit a number of deviations from their hardware specifications. In some cases, these deviations have been compensated for in the physical device driver design. Some of these deviations, however, cannot be resolved in software. The user must be familiar with the limitations and restrictions associated with such hardware.

When data is given to the transmit hardware, it has not yet been physically transmitted (at the RS232 interface). The data is considered completely transmitted by the transmit hardware at the physical RS232 interface when the transmit shift register of the UART is empty. The IOCtl ASYNC_GETLINESTATUS can be used to determine this information.

The device driver *transmit* queue is a memory buffer between the operating system and the *transmit* hardware. The device driver *receive* queue is a memory buffer between the operating system and the receive hardware. Both are considered to be owned by the physical device driver because the physical device driver controls the data movement in and out of the transmit and receive queues. Algorithms for this data movement can change between releases of the physical device driver. Changes in the ASYNC hardware can cause changes in the data movement algorithms and external interfaces.

Data that applications send (made available by Write requests) is placed in the physical device driver transmit queue. When an interrupt occurs to tell the physical device driver that the hardware is ready for more data, the driver gives the transmit hardware more data from the transmit queue.

When an interrupt occurs to tell the physical device driver that the hardware has received data, that data is placed in the physical device driver receive queue. When the physical device driver gets a Read request (READ request packet) from the application, it fills the Read request from the receive queue.

At high bit rates, such as 19200 bits-per-second, a serial device supporting full-duplex asynchronous I/O can generate an interrupt every 260 microseconds (at 10 bits-per-character and one interrupt-per-character transmitted and received). This leads to excessive interrupt-time overhead in the multitasking, interrupt-driven, device driver.

To address this problem, serial devices with Extended Hardware Buffering capabilities (FIFO or First-In-First-Out buffers) have been developed. However, many serially-attached devices that support the RS232-C interface, have been designed to operate with specific protocols that assume the system processes all data I/O one character at a time. The ASYNC physical device driver employs a software mechanism that automatically controls parameters to utilize the Extended Hardware Buffering capability, while compatibly supporting devices that use existing ASYNC device driver protocols.

The Automatic Protocol Override (system default) mode for Extended Hardware Buffering support partially utilizes only these performance advantages, while remaining fully compatible with the behavior of existing ASYNC device driver protocols (for example, Input Sensitivity using DSR). Applications and subsystems can disable certain device driver default settings in order to fully use the Extended Hardware Buffering capabilities. This results in a significant reduction of serial device interrupt processing overhead, and greatly increases the aggregate bit rates that can be supported across multiple active COM ports.

The size of the receive and transmit queues are available from the following IOCtls:

- Query Number of Characters in Receive Queue (ASYNC_GETINQUECOUNT)
- Query Number of Characters in Transmit Queue (ASYNC_GETOUTQUECOUNT)

The physical device driver services each communications port independently. Requests issued to a given port have no effect on any other communications ports that the physical device driver might be servicing. The physical device driver processes READ and WRITE request packets independently for a given port. An application can be written to support simultaneous reception and transmission of data. In addition, the device driver can process an IOCtl request simultaneously with outstanding Read and Write requests.

The physical device driver does not schedule the processing of IOCtl requests. It processes the IOCtl request when received, regardless of what else it is doing. This can cause unexpected results if, for instance, the bit rate is modified while data reception or transmission is taking place. The application should issue only one IOCtl request at a time. If it issues another IOCtl request before the first IOCtl request is completed, the results are UNDEFINED.

The device driver queues multiple READ and WRITE request packets independently and always begins processing the READ request packets in the order they are received. It also begins processing the WRITE request packets in the order they are received.

Note: The operating system does not guarantee that file system requests will be delivered to a device driver in the order in which they are issued by an application. This means that a request by one thread can get blocked in the operating system, thus allowing a subsequent request by a different thread for the same function (for example, DosWrite) to pass through and arrive ahead of the first thread at the physical device driver. This is true for synchronous operations performed by multiple threads or asynchronous operations performed by the same thread.

Because of thread-priority considerations and the system dynamics, the order observed by the application of completing requests of the same type might not be the order in which they were received by the device driver. The physical device driver always keeps the data in the same order in which the READ and WRITE request packets (of the same type) were received. There is no ordering or timing between different types of request packets.

The concept of a *First Level Open* is described in the section on States of the ASYNC Device Driver. A First Level Open occurs when the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close). A CLOSE request packet causing the physical device driver to process the next OPEN request packet as a First Level Open is called a *Last Level Close* . Because the requests that an application issues sometimes get out of order before they reach the device driver, an application cannot consider a close a Last Level Close until the CLOSE completes. If the application issues an Open request to the COM port before a previously issued Close request is completed, then the results are UNDEFINED.

A Flush request can be completed before all the appropriate request packets (which have been queued by the device driver) have been flushed. The appropriate request packets eventually are flushed and return to the caller, based on their priority and the system dynamics. Once the Flush request has been processed, the appropriate request packets do not cause data to be incorrectly transmitted (or received data to be moved).

The device driver supports different timeout processing characteristics and timeout settings for the Read and Write requests. Only the physical device driver is informed of when a given character is being transmitted or received at the hardware interface. Therefore, an application cannot expect to provide real-time flow control of data (in the middle of data transmission or reception) based on logical characters (XON/XOFF), or based on the state of the modem control signals by manually:

- Controlling or monitoring those modem control signals
- Monitoring the queue status
- Monitoring data moving across the link

Alternatively, the physical device driver provides optional modes of operation to control the data flow through the RS232-C port automatically. OS/2 applications use IOCtls to select which protocols are to be made active.

Output Modem Control Signals

In addition to allowing the application to directly control RTS and DTR, the physical device driver has different automatic control modes to control the value of the output modem control signals:

- Open and Close processing of DTR and RTS
- Disable/Enable DTR and RTS
- RTS toggling on transmit
- Input handshaking using DTR and RTS

These control modes are described in the section on States of the ASYNC Device Driver, and in the IOCtls description.

Note: The level of support provided by this device driver requires that DTR and RTS are turned on at least once, even if this puts the physical device driver in a mode where they will never be turned on again.

Input Modem Control Signals

Besides allowing the application to read directly the current state of DSR, CTS, DCD, and RI, the physical device driver has automatic modes that cause it to respond to the value that some input modem control signals can have:

- Output handshaking using CTS, DSR, DCD
- Input sensitivity using DSR

These control modes are described in the section on States of the ASYNC Device Driver and in the IOCtls description. Additional information on the state of the input modem control signals is available by using the IOCtl ASYNC_GETCOMMEVENT.

Logical Flow Control (XON/XOFF)

The application can attempt to manually control the flow of data by using the following IOCtls:

- Transmit Immediate (ASYNC_TRANSMITIMM)
- Stop Transmit Behave as if XOFF Received (ASYNC_STOPTRANSMIT)
- Start Transmit Behave as if XON Received (ASYNC_STARTTRANSMIT)

The physical device driver automatically controls the flow of transmitted data based upon the reception of XON/XOFF characters. This is referred to as Automatic Transmit Flow Control (XON/XOFF). The physical device driver also attempts to control the flow of data that is received by automatically transmitting XON/XOFF characters to the system it is communicating with, based on the amount of space left in the receive queue. This is referred to as Automatic Receive Flow Control (XON/XOFF).

Support for Extended Hardware Buffering

Another significant feature of this device driver is its exploitation of the Extended Hardware Buffering capabilities of the serial communications devices in many IBM systems and option adapters. Extended Hardware Buffering refers to the ability of the serial device servicing a COM port to buffer in hardware several characters, and to release them all at one time on the occurrence of a single transmit or receive hardware interrupt. This capability significantly reduces the interrupt-driven I/O processing overhead required to service Transmit and Receive requests on a given COM port. On the devices that support the Extended Hardware Buffering capability, this significantly improves COM I/O throughput and improves data integrity for higher data-transfer rates.

The Extended Hardware Buffering capabilities are automatically controlled under the default modes of the physical ASYNC device driver. Automatic Protocol Override is a feature of the OS/2 ASYNC device driver that automatically controls parameters relating to Extended Hardware Buffering. Systems and Adapters that incorporate the *FIFO-mode* hardware feature in a manner fully compatible with the NS-16550A UART are automatically enabled to run in Automatic Protocol Override mode.

Line Characteristics

IOCtls can be used to control and read the bit rate, number of stop-bits per character, number of data bits per character, and the parity characteristics of the line. See States of the ASYNC Device Driver.

Break and Error Processing

The device driver can be commanded to transmit a Break with an IOCtl (See IOCtls ASYNC_SETBREAKON and ASYNC_SETBREAKOFF).

An application can detect where an error or break occurred in the input data stream by using Break Replacement Character Processing and Error Replacement Character Processing. This requires that certain binary byte combinations be reserved for this purpose.

State of the COM Port

The following IOCtls can be used to determine the state of the COM port or if a given event happened. However, the exact timing relationship between this information and the specific data being received or transmitted at the time of the event is not available.

- Query COM Event Information (ASYNC_GETCOMMEVENT)
- Query COM Status (ASYNC_GETCOMMSTATUS)
- Query COM Error (ASYNC_GETCOMMERROR)

Event Notification

The device driver does not provide any capabilities of event notification. For example, the only way for an application to know that RI changed state or that a Break condition occurred is to poll that status with the IOCtl ASYNC_GETCOMMEVENT. This should not be a problem for those applications that can use the automatic control modes of the physical device driver during the course of a communications dialog (for time-critical control functions). Polling could be adequate for non-time-critical event monitoring.

Error Alert Generation

The ASYNC physical device driver supports SNA Generic Alerts by generating Error Alerts, as defined under the OS/2 Logging Facility. Alerts are generated by the ASYNC driver whenever the OS/2 Logging Facility is enabled by the user at system initialization time.

Alerts may be generated only while the COM port is open and is processing a Write request (transmitting data). Write Timeout mode must be normal. (If Infinite Timeout mode is enabled, timeouts do not occur.) Error Alerts can be generated only when a Write Timeout occurs while waiting for:

- CTS to be asserted, when Transmit is disabled because CTS is inactive. The Output Handshaking Using CTS mode must be enabled for this alert-generating condition to occur. This mode is enabled by default in the physical ASYNC device driver.
- DSR to be asserted, when Transmit is disabled because DSR is inactive. The Output Handshaking Using DSR mode must be enabled for this alert-generating condition to occur. This mode is enabled by default in the physical ASYNC device driver.
- DCD to be asserted, when Transmit is disabled because DCD is inactive. The Output Handshaking Using DCD mode must be enabled for this alert-generating condition to occur. This mode must be enabled by an application. It is not enabled by default in the ASYNC device driver.
- An XON to be received, when Transmit is disabled because an XOFF is received. Automatic Transmit Flow Control mode must be enabled for this alert-generating condition to occur. This mode must be enabled by an application. It is not enabled by default in the ASYNC device driver.

See the Set Device Control Block (DCB) Parameters Note on the IOCtl ASYNC_SETDCBINFO.

States of the ASYNC Device Driver

The different processing states of the physical ASYNC device driver, the ASYNC hardware, and the ASYNC control signals are listed below:

- Automatic Receive Flow Control (XON/XOFF)
- Automatic Transmit Flow Control (XON/XOFF)
- Bit Rate
- Break Replacement Character
- Break Replacement Character Processing
- COM Event WORD and COM Error WORD

- Data Bits
- DTR and RTS
- DTR Control Mode
- Error Replacement Character
- Error Replacement Character Processing
- Extended Hardware Buffering
- Input Sensitivity Using DSR
- Null Stripping
- Output Handshaking Using CTS, DSR, DCD
- Parity
- RTS Control Mode
- Read Timeout State
- Read Timeout Value
- Stop Bits
- Transmit Immediate
- Transmitting Break
- Write Timeout State
- Write Timeout Value
- XON/XOFF Characters

Each of the above states are covered as follows:

- A brief description
- The initial (default) value
- The effect on the physical device driver when the physical device driver receives an OPEN request packet for the port and the port is not already open (from a previous OPEN without a matching CLOSE)

 This is called a First Level Open. If applicable, the way the state of the physical device driver is affected by a CLOSE request packet.
- How the MODE utility can be used to alter the state of this item or how the MODE utility will alter the state of this item
- The effect on the physical device driver of the state of Extended Hardware Buffering

 In particular, special considerations relating to Automatic Protocol Override are given where the protocol being described is affected by this feature of the physical ASYNC device driver.

Automatic Receive Flow Control (XON/XOFF)

When the physical device driver is enabled for this mode of operation, it transmits an XOFF when its receive queue gets close to full, and an XON when its receive queue is about half full.

The normal mode of Automatic Receive Flow Control causes the ASYNC device driver to stop transmitting all data after it sends an XOFF character until its receive queue lowers to about half full, when it sends the XON character. This behavior is suitable for most devices, but reduces transmit throughput significantly in Full-Duplex (Transmit and Receive) communications scenarios. By setting the Full-Duplex mode of Automatic Receive Flow Control, the physical ASYNC device driver continues to send application data after it sends the XOFF character. See IOCtl ASYNC_SETDCBINFO.

Initial Value	Automatic Receive Flow Control is disabled.
First Level Open	There is no effect on whether the physical device driver is enabled or disabled for this mode of operation. The state of the physical device driver is reset to show that the last flow control character automatically transmitted was an XON, if it is enabled for this mode of operation.
Close Considerations	If the last character automatically transmitted by the physical device driver was an XOFF and a CLOSE request packet is received, the physical device driver automatically transmits an XON, if possible. After processing this close request, the port is not open any more from another open without a close.
Mode Utility	Always disables Automatic Receive Flow Control.

Automatic Transmit Flow Control (XON/XOFF)

When the physical device driver is enabled for this mode of operation, it stops sending data to the transmit hardware when an XOFF is received, and resumes sending data to the transmit hardware when an XON is received.

If this mode is enabled, Error Alerts can be generated when the OS/2 Logging Facility is enabled. If an external device sends an XOFF, but does not send an XON, transmission of data is blocked because the device driver is waiting for the XON to be received. If the XON is not received before the Write Timeout period expires, an Error Alert is generated.

Initial Value	Automatic Transmit Flow Control is disabled.
First Level Open	There is no effect on whether the physical device driver is enabled or disabled for this mode of operation. The state of the physical device driver is reset to show that it has not received an XOFF, so it can transmit (due to automatic transmit flow control), if it is enabled for this mode of operation.
Mode Utility	User interface to enable/disable this mode of the physical device driver.
Automatic Override	When Automatic Transmit Flow Control is enabled, the physical device driver responds to receiving the XOFF within a single character time. That is, Automatic Protocol Override controls the Extended Hardware Buffering capability so that only one character is buffered for transmit at a time, and the device generates an interrupt for every character received (Receive Trigger Level is set to 1).

Bit Rate

The bit rate determines the hardware setting for the data transfer rate, specified in bits per second, and is the speed for which the hardware is configured. See IOCtls ASYNC_SETBAUDRATE and ASYNC_GETBAUDRATE.

Initial Value	1200 bps
First Level Open	No effect
Mode Utility	User interface to change the bit rate

Break Replacement Character

The device driver uses this character value if Break Replacement Character Processing is enabled. See IOCtl ASYNC_SETDCBINFO.

Initial Value	00h
First Level Open	Reset to 00h
Mode Utility	No effect

Break Replacement Character Processing

If Break Replacement Character Processing is enabled and the device driver detects a break condition, it places the break replacement character in the physical device driver receive queue. If Break Replacement Character Processing is disabled, the physical device driver does not place any character in the physical device driver receive queue when it detects a break condition.

Initial Value	Break replacement character processing is disabled
First Level Open	Break replacement character processing is disabled
Mode Utility	No effect

COM Event WORD and COM Error WORD

These two WORDs have bits that show the status of the COM port. When an event occurs, the appropriate bits are turned on. The bits are cleared when the WORD is read with the appropriate IOCtl. See IOCtls ASYNC_GETCOMMEVENT and ASYNC_GETCOMMERROR.

Initial Value	All defined bits are 0
First Level Open	All defined bits are 0
Mode Utility	Not applicable

Data Bits

This is the number of bits contained in each character transmitted or received by way of the communications hardware. See IOCtls ASYNC_SETLINECTRL and ASYNC_GETLINECTRL.

Initial Value	7 data bits
First Level Open	No effect
Mode Utility	User interface to change the number of data bits

DTR and RTS

This is the value of the modem control signals Data Terminal Ready (DTR) and Request To Send (RTS) put out by the communications hardware. Each signal is controlled independently and can be either ON or OFF. See IOCtls ASYNC_SETMODEMCTRL and ASYNC_GETMODEMOUTPUT.

Initial Value	When the physical device driver starts the port during device driver initialization, their values are turned OFF.
First Level Open	The signals are normally turned ON, but there are many conditions that can cause the signals to be affected differently. See IOCtls ASYNC_SETMODEMCTRL and ASYNC_SETDCBINFO for a complete explanation.
Close Considerations	A Close request packet causes DTR and RTS to be turned OFF after the transmit hardware has completely transmitted all the data sent by the physical device driver. After processing this Close request, the port is no longer open from another OPEN without a CLOSE. In addition, at least 10 additional character times must have elapsed (or one second, whichever is less).
Mode Utility	Not applicable for direct control. Indirect effects through altering processing modes of the physical device driver are possible.

DTR Control Mode

The control modes for DTR are:

- Enable
- Disable
- Input Handshaking

The Enable and Disable control modes of DTR affect DTR processing during a First Level Open. When these control modes are set through IOCtl ASYNC_SETDCBINFO, the value of the DTR signal can be modified immediately by the physical device driver. The action depends on the previous control mode of DTR and the current value of the DTR modem control signal. If the control mode of DTR is Input Handshaking, then the device driver controls the DTR signal, depending on how full the receive queue is. The bits that control these states of the device driver are in the device control block.

Initial Value	Enable
First Level Open	No effect
Mode Utility	User interface to change the DTR Control Mode of the physical device driver

Error Replacement Character

The character value that the physical device driver uses, if Error Replacement Character Processing is enabled.

Initial Value	00h
First Level Open	Reset to 00h
Mode Utility	No effect

Error Replacement Character Processing

The processing that the physical device driver performs when a received character has an error (parity, framing, overrun, or lack of receive queue space) is determined by whether Error Replacement Character Processing is enabled (active).

Initial Value	Error replacement character processing is disabled
First Level Open	Error replacement character processing is disabled
Mode Utility	No effect

Extended Hardware Buffering

The extended hardware buffering (FIFO-mode) capabilities available in supported systems apply to the NS-16550A UART device and other fully compatible devices. These serial devices are installed on many IBM PS/2 system boards, and on various ASYNC communications adapter options. Refer to Hardware Support for Extended Hardware Buffering.

On those systems that incorporate serial devices that fully and compatibly support Extended Hardware Buffering, the OS/2 ASYNC device driver provides three modes for exploiting this feature:

- Enabled
- Disabled
- Automatic Protocol Override

The default is to enable Automatic Protocol Override on that COM port. Automatic Protocol Override is an asynchronous device driver mode of operation that automatically controls various parameters of Extended Hardware Buffering, such as Receive Trigger Level and Transmit Buffer Load Count.

Automatic Protocol Override causes the Receive Trigger Level and Transmit Buffer Load Count to be adjusted according to the device driver handshaking protocols in effect. When Automatic Protocol Override mode is ON and the handshaking protocols are set to their default settings, the physical device driver partially exploits only the performance advantages of Extended Hardware Buffering. The default handshaking protocols are:

- Enabled for Input Sensitivity Using DSR
- Enabled for Output Handshaking Using CTS and DSR
- Disabled for Output Handshaking Using DCD
- Disabled for Automatic Transmit Flow Control

If both Input Sensitivity Using DSR and Output Handshaking Using CTS and DSR are disabled, the Automatic Protocol Override causes the asynchronous device driver to automatically reset internal parameters (fully exploiting the Extended Hardware Buffering capabilities to the maximum extent possible).

The physical device driver's initialization default is to set Extended Hardware Buffering capabilities to the Automatic Protocol Override mode. An application or subsystem can alternatively set Extended Hardware Buffering to DISABLED, which causes the hardware to service transmit and receive interrupts one character at a time. It can also set Extended Hardware Buffering to ENABLED, which causes the physical device driver to set certain Extended Hardware Buffering parameters to specified levels, so the serial device fully exploits its Extended Hardware Buffering capabilities to the maximum extent possible.

When Extended Hardware Buffering is set to ENABLED, the following serial device hardware capabilities are exploited for maximum performance benefit and increased receive data integrity:

- By setting the Transmit Buffer Load Count to *16*, up to 16 characters are placed in the transmit hardware buffer (FIFO) during the processing of one *Transmit Holding Register Empty (THRE)* interrupt.

- A 16-character receive hardware buffer (FIFO) is available with the Receive Trigger Level set to *1, 4, 8*, or *14* characters. The Receive Trigger Level determines when the receive hardware generates a *Receive Data Available* hardware interrupt.

If the physical device driver receives an Open request for a COM port that is not already open, it does not alter the Extended Hardware

Buffering setting that is in effect at the time. The ASYNC physical device driver preserves this state across multiple Open and Close requests.

Initial Value	Automatic Protocol Override mode is on
First Level Open	No effect
Mode Utility	User interface to set the Extended Hardware Buffering mode to ENABLED, DISABLED, or to Automatic Protocol Override

Input Sensitivity Using DSR

When the physical device driver is enabled for this mode of operation and DSR is OFF, the physical device driver discards receive data.

Initial Value	Input Sensitivity using DSR is enabled
First Level Open	No effect
Mode Utility	User interface to enable/disable this mode of the physical device driver
Automatic Override	When Input Sensitivity Using DSR is enabled, the physical device driver responds to the lowering of the DSR line within a single character time. That is, the Extended Hardware Buffering (Receive Trigger Level) is adjusted so the device generates an interrupt for each character received. The full 16-character receive buffer is still available to help avoid any receive hardware overruns. The Transmit FIFO feature is also still enabled so that only one transmit interrupt occurs for every 16 characters transmitted. **Note:** In situations where DSR can naturally drop from O to OFF at the end of a communications session, but is not normally toggled during the session, it is most advantageous to disable Input Sensitivity Using DSR after the communications data transfer has begun. This achieves a significant performance benefit (under the control of Automatic Protocol Override), without the risk of losing valid data upon termination of the session. If, however, the DSR signal is expected to toggle frequently during a communications session, Input Sensitivity Using DSR should not be disabled, or *potential data loss can result* .

Null Stripping

If the physical device driver is enabled for null stripping, characters read in from the receive hardware (nonerror or nonbreak) with a value of 00h are thrown away. These null characters are stripped (not checked for Automatic Transmit Flow Control), even if the XON or XOFF character has been set to 00h, and are not placed in the physical device driver receive queue.

Initial Value	Null stripping is disabled
First Level Open	Null stripping is disabled
Mode Utility	No effect

Output Handshaking Using CTS, DSR, DCD

This mode of the physical device driver can be controlled independently for each modem control signal. When this mode is enabled, the physical device driver does not give data to the transmit hardware if the modem control signals are OFF (disabled). Data is not transmitted unless all the lines enabled for output handshaking are up.

If one of these modes is enabled, Error Alerts might be generated when the OS/2 Logging Facility is enabled. If an external device causes CTS, DCD, or DSR to become inactive, transmission is blocked while the device driver waits for the respective line to become active again. If the line does not become active before the Write Timeout period expires, an Error Alert is generated.

Initial Value	Output handshaking using CTS and DSR is enabled. Output handshaking using

	DCD is disabled.
First Level Open	No effect.
Mode Utility	User interface to enable/disable this mode of the physical device driver for CTS and DSR (independently). Mode always disables this mode of operation of the physical device driver for DCD.
Automatic Override	When Output Handshaking using DSR, CTS, or DCD is enabled, the physical device driver responds to the dropping modem line within a single character time. That is, Automatic Protocol Override controls the Extended Hardware Buffering capability so that only one character at a time is given to the transmit hardware (Transmit Buffer Load Count is set to 1). The Receive Trigger Level is unaffected.

Parity

Determines whether a parity bit exists and, if appropriate, what algorithm determines its value. See IOCtls ASYNC_SETLINECTRL and ASYNC_GETLINECTRL.

Initial Value	Even parity
First Level Open	No effect
Mode Utility	User interface to change the parity characteristics

RTS Control Mode

The control modes for RTS are:

- Enable
- Disable
- Input Handshaking
- Toggling on Transmit

The Enable and Disable control modes affect RTS processing during a First Level Open. When these control modes are set using IOCtl ASYNC_SETDCBINFO, the value of the RTS signal can be immediately modified by the physical device driver. The action depends on the previous control mode of RTS and the current value of the RTS modem control signal. If the control mode of RTS is Input Handshaking, the device driver controls the RTS signal, depending on how full the receive queue is. If the control mode of RTS is Toggling on Transmit, then the physical device driver controls the RTS signal, depending on whether it is transmitting data. The bits that control these states of the physical device driver are in the device control block.

Initial Value	Enable
First Level Open	No effect
Mode Utility	User interface to change the RTS Control Mode of the physical device driver

Read Timeout State

When the physical device driver processes a READ request packet, it can be with Normal, No-Wait, or Wait-For-Something Timeout processing. With *Normal* Timeout processing, if no data is received in the specified period of time, the request is completed. With *No-Wait* Timeout processing, the request obtains whatever data is available in the physical device driver receive queue (at the time the request is processed by the device driver) and returns. With *Wait-For-Something* Timeout processing, the request acts like No-Wait Timeout processing. However, if no data is available when the device driver processes the request, the physical device driver waits until some data is available or until the request times out due to Normal Timeout processing.

Initial Value	Normal Read Timeout processing
First Level Open	The device driver is set to Normal Read Timeout processing
Mode Utility	No effect

Read Timeout Value

The user-specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the Read Timeout processing, if *Normal Read Timeout* or *Wait For Something* Timeout processing is enabled.

Initial Value	1 minute
First Level Open	Set to 1 minute
Mode Utility	No effect

Stop Bits

Determines the number of stop bits associated with each character transmitted or received by way of the communications hardware.

Initial Value	1 stop bit
First Level Open	No effect
Mode Utility	User interface to change the number of stop bits

Transmit Immediate

The device driver can be told to transmit a byte immediately, bypassing the normal file system Write requests (bypassing the data to be transmitted in the transmit queue). Only one character at a time can be waiting to be transmitted *immediately* .

Initial Value	There is no character waiting to be transmitted immediately.
First Level Open	There is no character waiting to be transmitted immediately.
Close Considerations	A CLOSE request packet, when after processing this close request the port is not open any more (from another open without a close), causes the device driver to attempt to transmit the character waiting to be transmitted immediately. If the physical device driver cannot transmit the character waiting to be transmitted immediately (see IOCtl ASYNC_TRANSMITIMM), it does not try to transmit the character and proceeds with the close processing.
Mode Utility	Not applicable.

Transmitting Break

The device driver can be transmitting a break. See IOCtls ASYNC_SETBREAKON and ASYNC_SETBREAKOFF.

Initial Value	Not transmitting a break
Close Considerations	A CLOSE request packet, when after processing this close request the port is not open any more (from another open without a close), causes the device driver to tell the hardware not to transmit a break any more
Mode Utility	Not applicable

Write Timeout State

When the physical device driver processes a WRITE request packet, it can be with Normal or Infinite Timeout processing. With Normal Timeout processing if no data is given to the transmit hardware within a specified amount of time, the request is completed. With Infinite Timeout processing, the request is completed only when all the data from the request has been given to the transmit hardware.

Error Alerts can be generated on the occurrence of Write Timeout, if the OS/2 Logging Facility is enabled. In order for an error alert to be generated by the physical ASYNC device driver, the appropriate mode of operation must be enabled.

Initial Value	Normal Write Timeout processing
First Level Open	No effect on Write Timeout processing
Mode Utility	User interface to set Infinite or Normal Write Timeout processing

Write Timeout Value

The user-specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the Write Timeout processing, if Normal Write Timeout processing is enabled.

Initial Value	1 minute
First Level Open	Set to 1 minute
Mode Utility	No effect

XON/XOFF Characters

The characters used for automatic transmit and automatic receive flow control.

Initial Value	XON is 11h, and XOFF is 13h
First Level Open	The XON and XOFF characters are reset to their initial values
Mode Utility	No effect

Reserved Device Names (COM1-n)

The device name AUX does not appear in the device header of the ASYNC device driver. The ASYNC physical device driver does not support the reserved name AUX for either DOS applications or OS/2 applications.

IBM PS/2 (with Micro Channel) Considerations for COM1-4

The IBM PS/2 ASYNC device driver has device names COM1, COM2, COM3, and COM4 in its device driver header. Device name COM1 corresponds to the first LID in the ABIOS common data area with the ASYNC Device ID. Device name COM2 corresponds to the second LID. Device name COM3 corresponds to the third LID. Device name COM4 corresponds to the fourth LID in the ABIOS common data area with the ASYNC Device ID.

The Advanced BIOS architecture ensures that the ordering in the ROM BIOS 40: data area matches the ordering of the LIDs in the common data area. Compatibility BIOS supports up to four ASYNC devices on the IBM PS/2 system, and the physical device driver assumes that the order of the Logical IDs matches the order of the addresses of these devices in the ROM BIOS 40: data area. Additional ASYNC devices can be supported by additional device drivers.

The mapping of the ASYNC Logical ID ordering to COM*n* must be consistent across all device drivers that support the ASYNC hardware in the IBM PS/2 hardware environment.

Initialization/Resource Management

The device driver is loaded and initialized with a DEVICE= statement in the CONFIG.SYS file. They physical device driver processes parameters on the CONFIG.SYS to support COM ports with nonstandard address and IRQ's. Type **help com.sys** and read about the parameters.

During initialization, the physical device driver attempts to free memory from its data segment for ports it does not need and that do not get initialized. The physical device driver does not remove device names from its device driver header for ports that do not get initialized.

The device driver does not deinstall a device if the system requests it. If another device driver wishes to support a port already supported by this device driver, it needs to be initialized *before* this ASYNC device driver.

Shared ownership of a given serial device between multiple device drivers in a single session of the OS/2 operating system is supported, subject to certain restrictions. Each driver that installs sharing a serial device obtains exclusive access to that device when it processes an Open request, rather than claiming the device during initialization. Each driver also fully relinquishes control of that device, when it processes a matching Close request.

When the physical device driver is initialized, it is enabled by default for Output Handshaking Using CTS and DSR. This is for compatibility with existing systems (IBM PC and PS/2 BIOS INT 14H) and the requirements of supporting an RS232 port. It is also enabled by default for Input Sensitivity to DSR. This allows a remote device to indicate whether data being received is valid and is enabled to help ensure compatibility with existing systems. The initialization default for Extended Hardware Buffering on serial devices that fully support FIFO mode operations is Automatic Protocol Override.

Note: The ASYNC device driver default protocols provide an upwardly compatible RS232-C interface for communication with most devices. New communication applications and devices might not require the default protocols to be enabled. The user should evaluate these alternatives and consider which protocols to enable or disable in order to provide the highest possible performance for support of a given serial device.

Initialization Considerations

The device driver does not attempt to initialize or support a port if it does not get the INIT request packet for the port's corresponding device name. If the physical device driver gets the INIT request packet for a given device name, it checks to see if a valid I/O address is in the BIOS 40: data area that corresponds to that device name. COM1 is in 40:0 and COM2 is in 40:2. If the 40: area does not have a valid I/O address, the physical device driver fails the port and will not support the port. Otherwise, the device driver attempts to get exclusively the interrupt level that corresponds to the I/O address for the port. If the interrupt level is not available, the physical device driver fails to initialize the port and will not support the port.

If the interrupt level is available, the physical device driver relinquishes the interrupt level. The physical device driver also initializes the port and sets up support for the port during this startup of the operating system.

In summary, in order for the physical device driver to support a port, the following must be TRUE:

- The device driver must get an INIT request packet for the device name.
- The 40: area that corresponds to the device name must have a valid I/O address.
- The appropriate interrupt level must be available for exclusive use, even though the physical device driver will not claim the interrupt level for exclusive use during initialization.

The physical device driver claims ownership of the port by not deinstalling the corresponding device name. Another device driver can cause this device driver not to claim a port by initializing *before* this device driver, and by doing at least one of the following:

- Not allowing this physical device driver to receive an INIT request packet for a given device name
- Putting an invalid I/O address in the corresponding 40: area (for example, 0)
- Exclusively owning the appropriate interrupt level at initialization time

The device driver does not attempt to initialize or support a port if it does not get the INIT request packet for the port's corresponding device name. If the physical device driver gets the INIT request packet for a given device name, it attempts to claim ownership of the specific LID position for the ASYNC Device ID that corresponds to the device name being initialized.

For Micro Channel bus machines, if the LID is not available, the physical device driver fails to initialize the port and does not support the port. If the LID is available, the physical device driver initializes the port and sets up support for the port during this startup of the operating system. The LID for the port is relinquished; it is reclaimed during Open processing.

For the AT-bus machine, COM.SYS still installs even though the LID is not present.

In summary, for the physical device driver to support a port on an IBM PS/2 computer, the following must be TRUE:

- The physical device driver must get an INIT request packet for the device name.
- The ASYNC LID corresponding to the device name must be available.

The physical device driver claims ownership of the port by not deinstalling the corresponding device name. Another device driver can cause this device driver not to claim a port by initializing *before* this device driver and doing at least one of the following:

- Not allowing this device driver to receive an INIT request packet for a given device name
- Claiming the appropriate ASYNC LID

Data Translation/Monitor Support/Spooler Support

The physical device driver provides no data translation, code page, or monitor support. It is the responsibility of the application or subsystem to provide any function required in these areas.

Spooling from LPT to COM is supported by the spooler, but spooling from COM to LPT, or COM to COM, is not supported. When spooling from COM to LPT, code page support is provided by the spooler.

Any requests for registering or opening a monitor chain to COM*n* are rejected by the physical device driver. The physical device driver deals with binary data and provides no special processing of *binary* or *ASCII* data streams.

ASYNC Communication Device Driver Interfaces

The physical ASYNC device driver supports a large set of generic IOCtl functions (Category 01h), which are organized into the following groups:

- Break Processing
- Device Control Block (DCB) Parameter Access
- Device Driver I/O Queue Management
- Line Characteristics
- Manual XON/XOFF Processing
- Polled Event Functions

File System Requests

The physical ASYNC device driver supports the File System requests as shown in the following table.

Request	Description
INIT	Initialize the device
Open	Open the device
Close	Close the device
Read	Read from the device (Normal, Non-Destructive No-Wait)
Write	Write to the device
Status	Input or output status
Flush	Flush or terminate all pending requests

Open Processing

The physical device driver does not claim the interrupt level the port is on until the port is open. If the interrupt level is not available, the OPEN request packet fails. The interrupt level is claimed exclusively on the ISA bus machines. The interrupt level is claimed shareable on the Micro Channel architecture machines.

On PS/2 machines, a First Level Open causes the physical device driver to attempt to obtain a Logical ID. If this fails (which indicates another physical device driver might be using the device), a general failure is returned. If the Logical ID is obtained, but the open fails for some other reason, the Logical ID is freed. Other physical device drivers that coexist with the OS/2 physical ASYNC device driver perform similar processing.

If a timer tick handler is not available during First Level Open processing, the Open request can fail. If the physical device driver receives an OPEN request packet and the COM device is not already open (from a previous open without a close), this is called a *First Level Open* , and the physical device driver does special processing. See States of the ASYNC Device Driver. If a subsequent Open request is issued before a previous First Level Open request has completed, the device driver might process the OPEN request packets in an order different from the one in which they were issued. This could cause the First Level Open to take effect at a time different from what the application expected.

An Open request should never be issued until a previous Last Level Close request has completed. Otherwise, the function performed by a Last Level Close and a First Level Open might not occur. If the port is not already open (First Level Open), the physical device driver attempts to clear out any data in the *receive* hardware. On the IBM PS/2 system, if the port is not already open (First Level Open), the physical device driver relies on the *Reset/Initialize* Advanced BIOS function to reset and clear the UART receive hardware.

Close Processing

An application should never close an open handle to a COM port while there are requests still pending for that handle. If a request has not completed, it might be waiting for timeout processing. IOCtls are used to determine, and change, the current timeout processing.

A *Last Level Close* occurs when the port is no longer open (from another open without a close). When a Last Level Close occurs, the physical device driver does some special processing:

- Clears the receive and transmit queues
- Turns break OFF, if it is currently transmitting a break
- Clears any character waiting to be transmitted immediately, if it cannot be transmitted If it can be transmitted, the physical device driver makes sure that it is given to the transmit hardware
- Attempts to automatically transmit an XON (if possible), if currently enabled for Automatic Receive Flow Control (XON/XOFF), and the last character that the physical device driver automatically transmitted was an XOFF
- Waits until all the data in the transmit hardware has been physically transmitted
- Relinquishes the interrupt level
- Turns DTR and RTS OFF, if they are not already OFF. The physical device driver first waits the specified number of character times
- Relinquishes the Logical ID for the device (on PS/2 machines)

Read Processing

The physical device driver begins processing Read requests in the order it received them. Notice that this might not be the same order in which the requests were issued by the application. If the physical device driver receives more than one Read request, the request is queued on the Read request queue for later processing. Applications might not see Read requests completed in the order in which they were issued. The order of the data placed in the Read requests reflects the order in which the requests were received by the physical device driver.

The data for the Read requests comes from the physical device driver receive queue. Because of timeout processing, it is normal for the total number of Read characters requested not to be read. This is not considered an error. The request is completed when timeout is completed or when the amount of data requested is placed in the Read buffer. The various kinds of Read Timeout processing are discussed in the States of the ASYNC Device Driver. To reduce the probability of a device driver *receive* queue buffer overrun, the communications protocol takes into account the size of the physical device driver receive queue.

Write Processing

The physical device driver begins processing Write requests in the order it received them. Notice that this might not be the same order that the requests were issued by the application. If the physical device driver receives more than one Write request, the request is queued on the Write request queue for later processing. Applications might not see Write requests complete in the order they were issued. The order of the

data transmitted due to the Write requests reflects the order that the requests were received by the physical device driver.

The data from the Write requests is placed in the physical device driver transmit queue. The number of characters written is considered to be the number of characters given to the transmit hardware, and not the number of characters placed in the physical device driver transmit queue. Because of timeout processing, it is possible that the total number of Write characters requested will not be transmitted. This is not considered an error. The request is completed when it times out or when the amount of data requested is given to the transmit hardware (but not actually transmitted at the physical RS232-C interface). Write Timeout processing is discussed in States of the ASYNC Device Driver.

If infinite Write Timeout processing is enabled, it is the responsibility of the application to monitor the status of the Write requests. The application might have to issue an IOCtl to disable infinite Write Timeout processing to cause the Write request to be completed (without all the data being transmitted). If an application does not check that all the data is given to the transmit hardware on each Write request, use the infinite timeout processing mode of the physical device driver to ensure that all the data has been given to the *transmit* hardware before the request is completed. To increase the throughput (ratio of number of characters transmitted per second to the bit rate), the application should keep the Write requests as large as possible.

Access Authorization

The physical ASYNC device driver does not prevent multiple processes from concurrently opening the same device name. The physical device driver uses standard file system contention control. An application or subsystem that needs exclusive access to the COM device will open it with access rights set to DENY_ANY. Inheritance of open handles, and sharing of the device among multiple opens, works in a manner similar to regular files.

ASYNC (RS232-C) Generic IOCtl Command Summary

The following table lists each function and its description:

Function	Category 01h IOCtls
	Line Characteristics
41h	Set Bit Rate
42h	Set Line Characteristics (stop, parity, data bits)
43h	Extended Set Bit Rate
46h	Set Modem Control Signals
61h	Query Current Bit Rate
62h	Query Line Characteristics
63h	Extended Query Bit Rate
66h	Query Modem Control Output Signals
67h	Query Current Modem Input Signals
	Manual XON/XOFF (Flow Control) Processing
44h	Transmit Byte Immediate
47h	Behave as if XOFF Received (stop transmit)
48h	Behave as if XON Received (start transmit)
	Break Processing
45h	Set Break OFF
4Bh	Set Break ON
	Device Driver I/O Queue Management
68h	Query Number of Characters in Receive Queue
69h	Query Number of Characters in Transmit Queue

```
          Polled Events

64h          Query COM Status

65h          Query Transmit Data Status

6Dh          Query COM Error

72h          Query COM Event Information

          Enhanced Parameters

54h          Set Enhanced Mode Parameters

74h          Query Enhanced Mode Parameters

          Device Control Block (DCB) Parameter Access

53h          Set Device Control Block Parameters

73h          Query Device Control Block Parameters
```

Note: Device Control Block parameters determine:

- Automatic Transmit Flow Control (start/stop transmit, when XON/XOFF received)
- Automatic Receive Flow Control (transmit XON/XOFF, when receive buffer fills/empties)
- XON/XOFF Characters
- DTR Control Mode (enable/disable/input handshaking)
- RTS Control Mode (enable/disable/input handshaking/toggling on transmit)
- Output Handshaking Using CTS/DSR/DCD (control signal determines when to transmit)
- Input Sensitivity Using DSR (reception of data controlled by DSR)
- Error Replacement Character and Processing
- Break Replacement Character and Processing
- Null Stripping
- Timeout Processing
- Extended Hardware Buffering(DISABLED/ENABLED/Automatic Protocol Override)

See Generic IOCtl Commands for more information about Category 01h IOCtls.

DOS Session Considerations/Restrictions

Applications using a serial printer from a DOS Session must spool print data to the spooler through the appropriate LPT handles.

The physical device driver makes no attempt to restrict or mold the function of file system requests because they might have come from a DOS Session. To achieve the full capabilities of the file system access to COM, the application needs access to the full range of Category 01h IOCtls. The design and externals of the physical asynchronous device driver are based on the requirements of OS/2-mode applications that use the RS232-C port of the system.

Spooler Considerations

When setting up a serial printer in the Serial Port Settings, the user can choose between two handshaking protocols. If the user selects **None**, the serial printer card switches must be set for XON/XOFF handshaking. If the user chooses **Hardware**, the serial printer card switches must be set for DTR Pacing.

Performance

The achievable performance is very sensitive to the environment. The type and amount of other system activity determine the achievable performance. On the IBM PS/2 system, the number of COM ports or other devices on the same interrupt level significantly affects the achievable performance level.

Trying to receive data at too high a bit rate could cause *hardware overrun* errors or *receive queue overrun* errors. Receive queue overrun errors are easily solved by adjusting the communications protocol to the size of the physical device driver receive queue. Trying to transmit data at too high a bit rate could also cause the performance of the operating system to be severely reduced.

The bit rate can be set with the MODE command or with an IOCtl. The bit rate should not be set to values that might cause receive overruns or adverse OS/2 system performance effects.

Enabling Extended Hardware Buffering

In most transmit-only situations (for example, serial printers), there is always a requirement for flow control (using Output Handshaking or Automatic Transmit Flow Control). If the attached hardware can receive a significant number of characters after the modem control (pacing) signal is changed, then setting Extended Hardware Buffering to enabled (See IOCtl ASYNC_SETDCBINFO) can be an acceptable way to significantly improve the transmit throughput and the operating system performance. This allows the Extended Hardware Buffering to yield maximum serial I/O performance while still providing the required Output Handshaking or Automatic Transmit Flow Control protocols required by the attached serial device. Testing with Extended Hardware Buffering enabled must be performed at the attached device when the physical asynchronous device driver is placed in this mode.

In many Receive and Transmit (half- or full-duplex) scenarios, disabling Input Sensitivity Using DSR has no negative effects. Many communications devices have switches that cause DSR to be constantly ON. Disabling Input Sensitivity Using DSR significantly improves the ability of the serial port hardware to handle receive data without receive hardware overrun errors. Another possible approach is to set Extended Hardware Buffering enabled by using IOCtl ASYNC_SETDCBINFO or the OS/2 MODE command.

In some other Transmit and Receive scenarios, disabling Output Handshaking Using CTS and DSR after a communications link has been established has no adverse effects under normal operating conditions. Again,the achievable performance benefits can be significant. Another possible approach is to set Extended Hardware Buffering enabled by using IOCtl ASYNC_SETDCBINFO.

The potential negative effects of disabling a default control mode of the physical device driver should be thoroughly understood. The potential performance benefits, however, can far outweigh the added complexity of usage and any other potential problems. Such problems can usually be resolved either by reverting to the Automatic Protocol Override mode or by using IOCtl ASYNC_SETDCBINFO to set Extended Hardware Buffering to disabled.

The per-character processing requirements must be addressed when deciding whether to override one of the default protocols of the physical device driver. Possible data integrity problems, such as receive overrun errors, loss of data at the beginning or end of a communications session, or receipt of invalid data on a noisy communications connection can occur. Such problems must be considered before using the potential performance benefits associated with Extended Hardware Buffering.

For ports operating at a given data-transfer rate, the system has less than 20% interrupt-driven device driver overhead when running with Extended Hardware Buffering enabled (both transmit and receive FIFO buffering active), as compared with running those ports on devices where Extended Hardware Buffering is disabled.

Also of equal importance are the operational characteristics of the device driver. By setting Extended Hardware Buffering enabled, the physical device driver can transmit with the full 16-character FIFO buffer active (essentially transmitting 16 characters at a time), and the Receive Data Available interrupts can provide 4, 8, or 14 characters each to the physical device driver receive queue. This is true no matter what device driver protocols are enabled. Examples of scenarios where setting the FIFO Enabled mode of the physical device driver might be acceptable are:

- If the user does not care if too many characters are transmitted after a modem connection is broken
- If the printer or plotter connected to the system does not lose data when it tells the system (with a modem control signal), to stop transmitting, and the system continues to transmit a significant number (up to 16) of characters
- If the system is connected to a modem or another system that is normally set up to always keep DSR ON
- If the communications protocol with the remote device does not require the system to respond on a character-by-character basis to input data (for example, when the remote device sends data in blocks and provides *error retry* capability on a block basis rather than a per-character basis)

 Note: VCOM.SYS does not currently support buffering.

ASYNC_SETBAUDRATE (41h) - Set Bit Rate

ASYNC_SETBAUDRATE (41h) - Description

This function sets the bit rate.

ASYNC_SETBAUDRATE (41h) - Bit Rate

Bit Rate

The *Bit Rate* field is a binary integer representing the actual bit rate (bits-per-second) that the physical driver uses to set the bit rate of the COM device. For bit rates up to 19200 bps, the physical device driver sets the rate only if the hardware can support the rate within .01% margin of error. The exceptions are for 110 and 2000 bps, which can have an error of up to .026% and .69%, respectively. In all other cases, if the requested rate cannot be achieved by the hardware within .01% tolerance, the IOCtl fails with an ERROR_INVALID_PARAMETER return code.

For bit rates beyond 19200 bps, the physical device driver does not check the .01% margin of error. It is the user's responsibility to maintain the bit rate tolerance. After calling Function 41h to set a bit rate higher than 19200 bps, the user calls "Function 61h - Query Bit Rate" to check the actual bit rate set on a COM port.

The recommended bit rate values are:

- 110
- 150
- 300
- 600
- 1200
- 1800
- 2000
- 2400
- 3600
- 4800
- 7200
- 9600
- 19200
- 38400
- 57600

ASYNC_SETBAUDRATE (41h) - Parameter Packet Format

Field	Length	C Datatype
Bit Rate	WORD	USHORT

ASYNC_SETBAUDRATE (41h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_SETBAUDRATE (41h) - Returns

If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a general failure error is reported.

ASYNC_SETBAUDRATE (41h) - Remarks

If a general failure error is not returned, the physical device driver performs the action described in the *Bit Rate* field. An OPEN request packet does not cause the physical device driver to change the bit rate from its previous value. The initial value is 1200 bps.

The COM device hardware determines which bit rates can be supported on a given system. The range of bit rates supported by Function 41h is 2 bps to 19200 bps for COM ports on conventional serial devices. For COM ports on enhanced serial devices, the range is 10 bps to 57600 bps. A call with a value beyond this range fails with the ERROR_INVALID_PARAMETER return code.

ASYNC_SETBAUDRATE (41h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETBAUDRATE (41h)
Description:
Set Bit Rate

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_SETLINECTRL (42h) - Set Line Characteristics

ASYNC_SETLINECTRL (42h) - Description

This function sets the line characteristics (data bits, parity, stop bits).

ASYNC_SETLINECTRL (42h) - Data Bits

Data Bits
Has the following value and meaning:

00h-04h	Reserved
05h	5 data bits
06h	6 data bits
07h	7 data bits (initial value)
08h	8 data bits
09h-FFh	Reserved

ASYNC_SETLINECTRL (42h) - Parity

Parity

Has the following value and meaning:

00h	No parity
01h	Odd parity
02h	Even parity (initial value)
03h	Mark parity (parity bit always 1)
04h	Space parity (parity bit always 0)
05h-FFh	Reserved

ASYNC_SETLINECTRL (42h) - Stop Bits

Stop Bits

Has the following value and meaning:

00h	1 stop bit (initial value)
01h	1.5 stop bits (valid with 5-bit WORD length only)
02h	2 stop bits (not valid with 5-bit WORD length)
03h-FFh	Reserved

ASYNC_SETLINECTRL (42h) - Parameter Packet Format

Field	Length	C Datatype
Data Bits	BYTE	UCHAR
Parity	BYTE	UCHAR
Stop Bits	BYTE	UCHAR

ASYNC_SETLINECTRL (42h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_SETLINECTRL (42h) - Returns

If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure* error is reported and the line

characteristics are not changed for any parameters that were valid.

ASYNC_SETLINECTRL (42h) - Remarks

If a general failure error is not returned, the physical device driver sets the line characteristics as defined. An OPEN request packet does not cause the physical device driver to change the line characteristics from its previous values.

If the WORD length is less than 8 bits, the appropriate high-order bits for received data are set to 0 when placed in the receive queue and operated on by the physical device driver (for example, XON/XOFF checking, null stripping). This only applies to data that is received after the command is operated on by the physical device driver. Data already in the physical device driver receive queue is not affected in any way by a change in the WORD length.

If the WORD length is less than 8 bits, the physical device driver does not automatically truncate control/transmit data that is passed by the application. No error is reported by the device driver if transmit or control data given to it has high-order bits of non-zero value.

For example, if the physical device driver is told that the WORD length is 7 bits and the XOFF character is 80h, then a physical device driver that has Automatic Transmit Flow Control enabled is not able to recognize the XOFF character. If the error substitution character is set to 80h by the application, with a WORD length of 7 currently active, the device driver still places an 80h in the receive queue. It is the responsibility of the application to maintain consistency between the requested WORD length for the COM device and the requests that the application makes of the physical device driver.

ASYNC_SETLINECTRL (42h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETLINECTRL (42h)
Description:
Set Line Characteristics

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_EXTSETBAUDRATE (43h) - Extended Set Bit Rate

ASYNC_EXTSETBAUDRATE (43h) - Description

This function sets the bit rate in doublewords to cover bit rates higher than 19200 bps.

ASYNC_EXTSETBAUDRATE (43h) - Bit Rate

Bit Rate

The *Bit Rate* field is a binary integer representing the actual bit rate (bits-per-second) that the physical device driver uses to set the bit rate of the COM device. For a COM port that is supported by Enhanced UART or compatible serial communication devices, the range is 10 bps - 345600 bps. The user can get the minimum and maximum bit rates supported by the physical device driver on a COM port by using "Function 63h - Extended Query Bit Rate".

Note:

Due to system overhead and the physical limits on the hardware cables, actual bit rates might be restricted.

When this function is used for a COM port that has no support of the Enhanced UART or compatibles, then the bit range supported by the device driver is from 2 bps - 19200 bps. A call with a bit rate out of range fails with the ERROR_INVALID_PARAMETER return code.

ASYNC_EXTSETBAUDRATE (43h) - Fraction

Fraction

This field is a binary integer value that represents the fraction of the bit rate to set. It is used for setting a very precise bit rate. In most cases, this field is set to 0.

If the *Fraction* field needs to be set, the user must understand how bit rates are handled in the system. Bit rate is calculated using the following formula:

```
DIVISOR COUNT - BAUD CLOCK / SCALER / BINARY BIT RATE
```

For the Enhanced UART, the BAUD CLOCK is 22.1184 MHz and the SCALER is 32. The DIVISOR COUNT is a 16-bit value that is loaded into the ASYNC device's divisor latch to generate the final bit rate clock. Any fractional amount is rounded to the nearest divisor count to get the closest bit rate. This integer is put into the following formula to get the actual output bit rate value:

```
OUTPUT BIT RATE = BAUD CLOCK / SCALER / DIVISOR COUNT
```

For the value of the *Fraction* field, the resultant fraction obtained from the above formula is multiplied by 256. When setting a bit rate the physical device driver passes the bit rate and fraction directly to ABIOS.

For bit rates up to 19200 bps, the physical device driver sets the rate only if the hardware can support the rate within .01% margin of error to maintain compatibility. The exceptions are for 110 and 2000 bps, which can have an error of up to .026% and .69%, respectively. In all other cases, if the requested rate cannot be achieved by the hardware within .01% tolerance, the IOCtl fails with an ERROR_INVALID_PARAMETER return code.

For bit rates beyond 19200 bps, the physical device driver does not check the .01% margin of error. It is the user's responsibility to maintain the bit rate tolerance. After calling Function 43h, the user should call Function 63h to check the actual bit rate set on a COM port.

The recommended bit rate values are:

- 110
- 150
- 300
- 600
- 1200
- 1800
- 2000
- 2400
- 3600
- 4800
- 7200
- 9600
- 19200
- 38400
- 57600
- 76800 *
- 115200 *
- 230400 *
- 345600 *

*** Note:** Use IOCtl Function 74h.

ASYNC_EXTSETBAUDRATE (43h) - Parameter Packet Format

Field	Length	C Datatype
Bit Rate	DWORD	ULONG
Fraction	BYTE	UCHAR

ASYNC_EXTSETBAUDRATE (43h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_EXTSETBAUDRATE (43h) - Returns

If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure error* is reported.

ASYNC_EXTSETBAUDRATE (43h) - Remarks

If a general failure error is not returned, the physical device driver performs the action described in the Bit Rate field. An OPEN request packet does not cause the physical device driver to change the bit rate from its previous value. The initial value is 1200 bps.

The physical ASYNC device driver is designed to handle high bit rates under optimum conditions. To achieve successful data transfer operations at high bit rates and to obtain high data throughput, the following conditions must be met:

- The connecting cables must be free from electrical noise and must maintain the voltage conforming to the RS232-C standard. IBM Personal Computer* Communication adapter cables (P/N 6323741 or 1502067) can guarantee this for up to 20 feet.
- DMA capability must be available.
- There is little system overhead. Operations at high bit rates require many CPU cycles. It is important that no other tasks be active and that the ASYNC Communication application be written to reduce the CPU overhead as much as possible.

 ASYNC Communication applications should have optimal data block sizes for DosRead or DosWrite requests to the physical ASYNC device driver. In general, as block size increases, the overhead related to making those requests decreases. However, these applications might need more time to handle large data chunks for CRC checking or saving the data into a file as this takes CPU cycles away from the physical ASYNC device driver.
- Appropriate communication protocols might be required for the prevention of the physical ASYNC device driver receive buffer overflow.
- Applications should not use the Error or Break Replacement operations. If these options are taken, the physical ASYNC device driver runs in Data/Status mode and cannot take advantage of 80386 machine instructions, such as REP MOVED. The physical device driver then has to check each status byte, which slows down the performance.

This function is extended from ASYNC_SETBAUDRATE.

ASYNC_EXTSETBAUDRATE (43h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_EXTSETBAUDRATE (43h)
Description:
Extended Set Bit Rate

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_TRANSMITIMM (44h) - Transmit Byte Immediate

ASYNC_TRANSMITIMM (44h) - Description

This function transmits byte immediate.

ASYNC_TRANSMITIMM (44h) - Character to be Transmitted

Character to be Transmitted
None.

ASYNC_TRANSMITIMM (44h) - Parameter Packet Format

Field	Length	C Datatype
Character to be Transmitted	BYTE	UCHAR

ASYNC_TRANSMITIMM (44h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_TRANSMITIMM (44h) - Returns

If the call is made with an invalid Data Packet pointer or if there is already another character waiting to be transmitted immediately due to a previous Function 44h request that has not been fulfilled, then a *general failure* error is reported and this request is ignored. A Transmit Immediate request is considered fulfilled when the character is given to the transmit hardware.

ASYNC_TRANSMITIMM (44h) - Remarks

If a general failure is not returned, the physical device driver immediately transmits the byte contained in the Parameter Packet subject to the following conditions:

- If data in the transmit queue is currently being transmitted or waiting to be transmitted, the character to be transmitted immediately is placed at the logical front of the physical device driver transmit queue (not considered *in* the transmit queue), so that it is the next character to be given to the transmit hardware. If Automatic Receive Flow Control is enabled, then an XON or XOFF character can be placed ahead of the character to be transmitted immediately.

- This request is always completed immediately (before the character is actually transmitted) even if the character might not be immediately transmitted. If there already is one character waiting to transmit immediately (based on a previous request), then a general failure error is returned. The application must make this request again after there is no character (waiting to transmit immediately) in the device driver transmit queue. ASYNC_GETCOMMSTATUS can be used to determine whether a character is currently waiting to be transmitted immediately.

- The physical device driver does not immediately transmit the character (waiting to transmit immediately) if the physical device driver is not transmitting characters because of modem control signal output handshaking (see ASYNC_SETDCBINFO) or if the physical device driver is currently transmitting a break.

- If the physical device driver is not transmitting characters because of Automatic Transmit or Receive Flow Control (XON/XOFF) is enabled, or because it was asked to operate as if an XOFF character had been received (see ASYNC_STOPTRANSMIT), then the physical device driver still transmits a character that is waiting to be transmitted immediately due to this request.

 Warning: If Automatic Transmit or Receive Flow Control is enabled, an application may or may not request that the device driver transmit a character immediately. Otherwise, unexpected results can occur.

- The Transmit Byte Immediate IOCtl is generally used to manually send XON and XOFF characters.

- The character waiting to be transmitted immediately is not considered part of the physical device driver transmit queue and is not flushed as a flush request. XON/XOFF characters that are automatically transmitted as the result of Automatic Receive Flow Control might be placed ahead of the character waiting to be transmitted immediately. Applications should not have dependencies on this ordering.

- If the serial port controller fully supports Extended Hardware Buffering capabilities and the physical device driver is set with Extended Hardware Buffering enabled, then calling this function results in temporarily setting the Transmit Buffer Load Count to 1 (to load the transmit immediate byte). If the physical device driver conditions allow data to be transmitted, then the byte is sent and the device driver then resumes operations with the previously prevailing Transmit Buffer Load Count (as determined by Function 53h).

ASYNC_TRANSMITIMM (44h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_TRANSMITIMM (44h)
Description:
Transmit Byte Immediate

Select an item:
Description

Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_SETBREAKOFF (45h) - Set Break OFF

ASYNC_SETBREAKOFF (45h) - Description

This function sets the break *off*.

ASYNC_SETBREAKOFF (45h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_SETBREAKOFF (45h) - COM Error WORD (COMERR)

COM Error WORD (COMERR)
The physical device driver returns this information if a general failure error is not reported. See "Function 6Dh - Query COM Error" for COMERR definition. The COM device error information is not cleared by this action.

ASYNC_SETBREAKOFF (45h) - Data Packet Format

Field	Length	C Datatype
COM Error WORD (COMERR)	WORD	USHORT

ASYNC_SETBREAKOFF (45h) - Returns

If the call is made with an invalid Parameter Packet pointer and a *general failure* error is reported, this function is not performed and valid information is not returned in the Data Packet.

ASYNC_SETBREAKOFF (45h) - Remarks

If a general failure is not returned, then the physical device driver stops generating a break signal. It is not considered an error if the physical device driver is not generating a break signal. The physical device driver then resumes transmitting characters, taking into account all the other reasons why characters might be transmitted.

ASYNC_SETBREAKOFF (45h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETBREAKOFF (45h)
Description:
Set Break OFF

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_SETMODEMCTRL (46h) - Set Modem Control Signals

ASYNC_SETMODEMCTRL (46h) - Description

This function sets the modem control signals.

ASYNC_SETMODEMCTRL (46h) - Modem Control Signals ON Mas

Modem Control Signals ON Mask

The physical device driver sets the modem control signals as defined in this field. Bit 0 is DTR; bit 1 is RTS. If any other bits are set/reset by the masks, then a general failure error results. The OFF mask contains a mask (with all bits equal to 0) to turn the modem control signal *off*. The ON mask contains a mask (with all bits equal to 1) to turn the modem control signal *on*. If the Parameter Packet indicates both turn *off* and turn *on* on the same bit, the bit will be turned *on*.

For example:

Mask ON	Mask OFF	Description
01h	FFh	Set DTR
00h	FEh	Clear DTR
02h	FFh	Set RTS

00h	FDh	Clear RTS
03h	FFh	Set DTR and RTS
00h	FCh	Clear DTR and RTS

If DTR Control mode input handshaking, RTS Control mode input handshaking or *toggling on transmit* is set, then this request cannot try to change the state of the modem control signals, that are being used for input handshaking or toggling on transmit. If the request tries to modify a modem control signal that is being used for input handshaking or toggling on transmit, a *general failure* error results.

ASYNC_SETMODEMCTRL (46h) - Modem Control Signals OFF Ma

Modem Control Signals OFF Mask

See Modem Control Signals ON Mask above.

ASYNC_SETMODEMCTRL (46h) - Parameter Packet Format

Field	Length	C Datatype
Modem Control Signals ON Mask	BYTE	BYTE
Modem Control Signals OFF Mask	BYTE	BYTE

Related C Data Structure

The MODEMSTATUS data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

ASYNC_SETMODEMCTRL (46h) - COM Error WORD (COMERR)

COM Error WORD (COMERR)

The physical device driver returns this information if a general failure error is not returned to the application. See the ASYNC_GETCOMMERROR for COMERR definition. The COM device error information is not cleared by this action. At device driver initialization, the device driver turns *off* DTR and RTS for the COM devices that it owns.

When the COM device is not already open (from a previous Open without a Close), the OPEN request packet causes DTR and RTS to be set according to the DTR Control mode and the RTS Control mode. See Note 1 of ASYNC_SETDCBINFO.

Note: If the port will not be open after processing a CLOSE request packet (Last Level Close), DTR and RTS are turned *off* by the device driver. This occurs after the transmit hardware has completely transmitted (at the physical RS232 interface) all the data that it has been given by the physical device driver and time for at least 10 additional character transmissions has elapsed.

ASYNC_SETMODEMCTRL (46h) - Data Packet Format

Field	Length	C Datatype
COM Error WORD (COMERR)	WORD	USHORT

ASYNC_SETMODEMCTRL (46h) - Returns

If the call is made with invalid Parameter Packet values and a general failure error is reported, then the modem control signals are not changed and the Data Packet information returned to the application is *undefined*.

ASYNC_SETMODEMCTRL (46h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETMODEMCTRL (46h)
Description:
Set Modem Control Signals

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

ASYNC_STOPTRANSMIT (47h) - Behave as if XOFF Received

ASYNC_STOPTRANSMIT (47h) - Description

This function behaves as if an XOFF was received (stops transmitting).

ASYNC_STOPTRANSMIT (47h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_STOPTRANSMIT (47h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_STOPTRANSMIT (47h) - Returns

If the call is made with invalid Parameter Packet or Data Packet pointers, a *general failure* error is reported, and this request is not performed by the physical device driver.

ASYNC_STOPTRANSMIT (47h) - Remarks

If a *general failure* error is not returned by the physical device driver, this function causes data transmission to be stopped by preventing the physical device driver from sending additional data to the transmit hardware.

If Automatic Transmit Flow Control is enabled, this request causes the physical device driver to behave as if it had received the XOFF character. Transmission can be resumed when an XON is received by the physical device driver after a ASYNC_STARTTRANSMIT request is received, or when the physical device driver is told to disable Automatic Transmit Flow Control (if it was enabled).

If Automatic Transmit Flow Control is disabled, then the Function 48h request is required for transmission to be resumed. If, after this request is received, the physical device driver is told to enable Automatic Transmit Flow Control, then transmission is still disabled, but can be re-enabled. See Note 2 of ASYNC_SETDCBINFO.

See ASYNC_GETCOMMSTATUS for other reasons why transmission might be disabled.

ASYNC_STOPTRANSMIT (47h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_STOPTRANSMIT (47h)
Description:
Behave as if XOFF Received

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_STARTTRANSMIT (48h) - Behave as if XON Received

ASYNC_STARTTRANSMIT (48h) - Description

This function behaves as if an XON was received (starts transmitting).

ASYNC_STARTTRANSMIT (48h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_STARTTRANSMIT (48h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_STARTTRANSMIT (48h) - Returns

If the call is made with invalid Parameter Packet or Data Packet pointers, a general failure error is reported, and this request is not performed by the physical device driver.

ASYNC_STARTTRANSMIT (48h) - Remarks

If a general failure error is not returned by the physical device driver, this function allows data transmission to be resumed by the device driver if the data transmission was stopped because of a ASYNC_STOPTRANSMIT request, or because an XOFF character was received while the physical device driver was in Automatic Transmit Flow Control mode.

See ASYNC_GETCOMMSTATUS for other reasons why transmission might be disabled.

ASYNC_STARTTRANSMIT (48h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_STARTTRANSMIT (48h)
Description:
Behave as if XON Received

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_SETBREAKON (4Bh) - Set Break On

ASYNC_SETBREAKON (4Bh) - Description

This function sets break on.

ASYNC_SETBREAKON (4Bh) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_SETBREAKON (4Bh) - COM Error Word (COMERR)

COM Error Word (COMERR)

The physical device driver returns this information if a general failure error is not reported. See ASYNC_GETCOMMERROR for COMERR definition. The COM device error information is not cleared by this action.

If, after processing this Close request, the port will not be open (from another Open without a Close), a CLOSE request packet causes break to be turned *off*.

ASYNC_SETBREAKON (4Bh) - Data Packet Format

Field	Length	C Datatype
COM Error Word (COMERR)	WORD	USHORT

ASYNC_SETBREAKON (4Bh) - Returns

If the call is made with an invalid Parameter Packet pointer and a general failure error is reported, this function is not performed and valid information is not returned in the Data Packet.

ASYNC_SETBREAKON (4Bh) - Remarks

If a general failure error is not returned, the physical device driver generates the break signal immediately. It is not considered an error if the physical device driver is already generating a break signal. The device driver does not wait for the transmit hardware to become empty. Note that more data is not given to the transmit hardware until the break is turned *off*. The break signal is always transmitted, regardless of whether or not the physical device driver is transmitting characters for other reasons.

ASYNC_SETBREAKON (4Bh) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETBREAKON (4Bh)
Description:
Set Break On

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_SETDCBINFO (53h) - Set Device Control Block (DCB) Par

ASYNC_SETDCBINFO (53h) - Description

This function sets the Device Control Block (DCB) information.

ASYNC_SETDCBINFO (53h) - Write Timeout

Write Timeout
Specifies the time period used for Write Timeout processing. The value is in units of .01 seconds based on 0 (where 0=.01 seconds). See Note 8.

ASYNC_SETDCBINFO (53h) - Read Timeout

Read Timeout
Specifies the time period used for Read Timeout processing. The value is in units of .01 seconds based on 0 (where 0=.01 seconds). See Note 9.

ASYNC_SETDCBINFO (53h) - Flags1

Flags1
Has the following bits:

Bits 0-1 — DTR Control Mode. See Note 1.

Bit 1	Bit 0	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Invalid input, resulting in a general failure error.

Bit 2 — Reserved. Set to 0.

Bit 3 — Enable output handshaking using CTS. See Note 3.

Bit 4 — Enable output handshaking using DSR. See Note 3.

Bit 5 — Enable output handshaking using DCD. See Note 3.

Bit 6 — Enable input sensitivity using DSR. See Note 4.

Bit 7 — Reserved. Set to 0.

ASYNC_SETDCBINFO (53h) - Flags2

Flags2

Has the following bits:

Bit 0 — Enable Automatic Transmit Control Flow (XON/XOFF). See Note 2.

Bit 1 — Enable Automatic Receive Control Flow (XON/XOFF). See Note 2.

Bit 2 — Enable error replacement character. See Note 5.

Bit 3 — Enable null stripping. See Note 6.

Bit 4 — Enable break replacement character. See Note 7.

Bit 5 — Automatic Receive Flow Control:

0 = Normal
1 = Full Duplex.

Bits 6-7 — RTS Control Mode. See Note 1.

Bit 7	Bit 6	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Toggling on transmit

ASYNC_SETDCBINFO (53h) - Flags3

Flags3

Has the following bits:

Bit 0 — Enable Write Infinite Timeout processing. See Note 8.

Bits 1-2 — Enable Read Timeout processing. See Note 9.

Bit 2	Bit 1	Description
0	0	Invalid input. Results in a general failure error.
0	1	Normal Read Timeout processing
1	0	Wait-For-Something Read Timeout processing
1	1	No-Wait Read Timeout processing.

Bits 3-4 — Extended Hardware Buffering. See Note 10.

Bit 4	Bit 3	Description
0	0	Not supported; ignored. No change to FIFO state.
0	1	Disabled
1	0	Enabled
1	1	Automatic Protocol Override.

Bits 5-6 — Received trigger level. See Note 11.

Bit 6	Bit 5	Description
0	0	1 character
0	1	4 characters
1	0	8 characters
1	1	14 characters

Note: The trigger level must be set to 1 character when Enhanced mode is *on*.

Bit 7 — Transmit Buffer Load Count. See Note 12.

0 = 1 character
1 = 16 characters.

Note: When Extended Hardware Buffering is disabled, this bit is set by the device driver.

ASYNC_SETDCBINFO (53h) - Error Replacement Character

Error Replacement Character
Any byte value in the range 00h-FFh. See Note 5.

ASYNC_SETDCBINFO (53h) - Break Replacement Character

Break Replacement Character
Any byte value in the range 00h-FFh. See Note 7.

ASYNC_SETDCBINFO (53h) - XON Character

XON Character
Any byte value in the range 00h-FFh. See Note 2.

ASYNC_SETDCBINFO (53h) - XOFF Character

XOFF Character
Any byte value in the range 00h-FFh. See Note 2.

ASYNC_SETDCBINFO (53h) - Parameter Packet Format

Field	Length	C Datatype
Write Timeout	WORD	USHORT
Read Timeout	WORD	USHORT
Flags1	BYTE	BYTE
Flags2	BYTE	BYTE
Flags3	BYTE	BYTE
Error Replacement Character	BYTE	BYTE
Break Replacement Character	BYTE	BYTE
XON Character	BYTE	BYTE
XOFF Character	BYTE	BYTE

Related C Data Structure

The DCBINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

ASYNC_SETDCBINFO (53h) - Data Packet Format

None. Packet pointer must be NULL.

ASYNC_SETDCBINFO (53h) - Returns

If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure* error is reported and none of the Device Control Block (DCB) characteristics of the physical device driver for this COM device are changed.

ASYNC_SETDCBINFO (53h) - Remarks

If a general failure error is not returned, then the actions described below are taken by the physical device driver. Notice that all reserved bit fields that are specified as *Set to 0* must be set to zero on entry to the physical device driver, or a general failure error results. The same applies to the NULL Data Packet pointer.

The general DCB parameter access functions, ASYNC_SETDCBINFO and ASYNC_GETDCBINFO are used:

- For Automatic Transmit Flow Control (start/stop transmit, when XON/XOFF character received)
- For Automatic Receive Flow Control (transmit XON/XOFF, when receive buffer fills or empties)
- To determine XON/XOFF characters
- For DTR Control mode (enable/disable/input handshaking)
- For RTS Control mode (enable/disable/input handshaking/toggling on transmit)
- For output handshaking using CTS/DSR/DCD (control signal determines when to transmit)

- For input sensitivity using DSR (reception of data controlled by DSR)
- For error replacement character and processing
- For break replacement character and processing
- For null stripping
- To Receive or Transmit Timeout processing

To maintain upward compatibility, the application should call ASYNC_GETDCBINFO before Function 53h is used. This allows the *reserved* bits to be set correctly in a future release of the physical device driver. By doing the return first, the application maintains the state of the physical device driver for a mode that the application is not aware of.

Note 1

Control of DTR and RTS. The physical device driver allows the caller to automatically control the setting of Data Terminal Ready (DTR) and Request To Send (RTS) through the RTS Control mode and the DTR Control mode settings of ASYNC_SETDCBINFO. The application can also request manual control over these modem control signals. The ways in which these signals can be controlled are as follows:

Set RTS Control Mode to Toggling on Transmit: If the Flags2 bits 7, 6 are set to 1, then the physical device driver is in the automatic control mode of RTS. When the physical device driver is initialized, the RTS Control mode is enabled; therefore, initially the device driver is not in the automatic control mode of RTS. Notice that this mode of operation of the physical device driver should be enabled only when the system is attached to devices that do not present data to the system receive hardware when RTS is *on*. In this mode, the device driver:

- Always turns on RTS, if a break is being transmitted.
- Does not turn RTS *off* until the transmit hardware has emptied its buffers, once data is in the transmit hardware buffer.
- Turns on RTS, if not already *on*, when there is data in the physical device driver transmit queue or when there is an outstanding WRITE request packet. The following are two exceptions:
 - The physical device driver is allowed to transmit, even if Automatic Transmit/Receive Flow Control (XON/XOFF) is enabled. The physical device driver needs to turn on RTS momentarily to transmit a character immediately, if not normally allowed to transmit due to Automatic Transmit/Receive Flow Control.
 - The physical device driver is allowed to transmit because it was not told to behave as if an XOFF had been received (Function 47h). The physical device driver needs to turn on RTS momentarily to transmit a character immediate, if not normally transmitting due to XOFF flow control considerations. Also, the physical device driver needs to turn on RTS momentarily to transmit an XON or XOFF due to Automatic Receive Flow Control, if not normally transmitting due to XOFF flow control considerations.
- Turns off RTS, if not already *off*, when one of the following conditions is TRUE:
 - There is no data in the physical device driver transmit queue (and no data in Write requests in progress), there are no queued Write requests, and the transmit hardware has physically transmitted (at the physical RS232 interface) all the data that it has been given.
 - The physical device driver is not allowed to transmit because Automatic Transmit/Receive Flow Control (XON/XOFF) is enabled or because the driver was asked to behave as if an XOFF had been received (Function 47h). The physical device driver needs to turn on RTS to transmit a character immediately or XON/XOFF, because of Automatic Receive Flow Control (XON/XOFF). RTS is never turned *off* until the transmit hardware has physically transmitted all the data that it has been given.
- When this function is enabled, the physical device driver controls RTS, as determined by the above description.
- If this function is disabled because a new RTS Control mode is chosen, then the new RTS Control mode determines the state of the RTS.
- The device driver does not examine any other modem control signals before it turns RTS *off* or *on*.

An OPEN request packet does not cause the physical device driver to change its RTS Control mode. The physical device driver maintains the state of this mode of operation across OPEN request packets. When the physical device driver is in the RTS Control mode *toggling on transmit*, then it does not allow the application to control RTS by using ASYNC_SETMODEMCTRL.

Set DTR and RTS Control Mode to Input Handshaking: When the Flags1 bits 1, 0 are set to 1, 0, the DTR Control mode is set to input handshaking. Setting bits 7, 6 of Flags2 to 1, 0 sets the RTS Control mode to input handshaking. When the physical device driver is initialized, the RTS and DTR Control modes are enabled; so initially the device driver is not in the automatic control mode of RTS and DTR. Notice that this mode of operation of the physical device driver is set only when there is the possibility of a physical device driver receive queue overrun, and the system is attached to data terminal equipment that stops transmitting data when the appropriate modem control signals are turned *off*.

Because the Input Handshaking mode can be set for RTS, DTR, or both, the DTR and RTS Control modes are processed independently. In Input Handshaking mode, the physical device driver:

- Turns the appropriate modem control signals *on* when the device driver receive queue is less than half-full
- Turns the appropriate modem control signals *off* when the device driver receive queue is almost full
- Does not monitor the value of the appropriate modem control signals (when this mode is first set and the queue size is between half-full and almost full)
- Determines the correct value of the modem control signals when this function is enabled, and controls them accordingly.

If this function is disabled by choosing a new RTS or DTR Control mode, then the new control mode determines the state of the RTS or DTR.

- Does not examine any other modem control signals before controlling DTR or RTS

An OPEN request packet does not cause the physical device driver to change its RTS and DTR Control modes. The driver maintains the state of these modes of operation across OPEN request packets. When the physical device driver is in the RTS Control mode input handshaking, it does not allow the application to control RTS through Function 46h. When the physical device driver is in the DTR Control mode input handshaking, it does not allow the application to control DTR through Function 46h.

Set DTR and RTS Control Mode to Enable or Disable: OPEN processing has the following settings:

- Flags1 bits 1, 0 are set to 0, 0. DTR Control mode is disabled.
- Flags1 bits 1, 0 are set to 0, 1. DTR Control mode is enabled.
- Flags2 bits 7, 6 are set to 0, 0. RTS Control mode is disabled.
- Flags2 bits 7, 6 are set to 0, 1. RTS Control mode is enabled.

When the physical device driver is initialized, the RTS and DTR Control modes are enabled, but the value of the modem control signals is *off* until the port gets an OPEN request packet. An OPEN request packet does not cause the physical device driver to change its RTS and DTR Control modes. The driver maintains the state of these modes of operation across OPEN request packets.

Because Enable or Disable modes can be set for either RTS, DTR, or both, the DTR and RTS Control modes are processed independently. If the RTS Control mode is disabled when the physical device driver receives an OPEN request packet, and the device is not already open (from a First Level Open), the RTS modem control signal is kept (turned) *off* during the OPEN processing. If the RTS Control mode is enabled when the physical device driver receives an OPEN request packet, and the device is not already open, the RTS modem control signal is turned *on* during the OPEN processing.

If the RTS Control mode is set to disable and the previous mode was not disable, the RTS modem control signal is turned *off*. If the RTS Control mode is set to disable and the previous mode was also disable, this IOCtl has no effect on the RTS modem control signal.

If the RTS Control mode is set to enable and the previous mode was not enable the RTS modem control signal is turned *on*. If the RTS Control mode is set to enable and the previous mode was also enable, this IOCtl has no effect on the RTS modem control signal.

The following tables summarize the above information:

From Control Mode	To Control Mode	Effect on Control Signal
Disable	Disable	None
Disable	Enable	Turn ON
Disable	Input handshaking	Modem control signal controlled automatically
Enable	Disable	Turn OFF
Enable	Enable	None
Enable	Input handshaking	Modem control signal controlled automatically
Input handshaking	Disable	Turn OFF
Input handshaking	Enable	Turn ON
Input handshaking	Input handshaking	Modem control signal controlled automatically

From Control Mode	To Control Mode	Effect on Control Signal
Disable	Toggle on transmit	Modem control signal controlled automatically
Enable	Toggle on transmit	Modem control signal controlled automatically

Input handshaking	Disable	Modem control signal controlled automatically
Toggle on transmit	Disable	Turn OFF
Toggle on transmit	Enable	Turn ON
Toggle on transmit	Input handshaking	Modem control signal controlled automatically
Toggle on transmit	Toggle on transmit	Modem control signal controlled automatically

Because the initial Control mode of the physical device driver is enabled for RTS and DTR, both modem control signals are turned *on* when the port is first opened. If the physical device driver receives an OPEN request packet, and the device is already open, the physical device driver does not alter the value of the RTS and DTR modem control signals, regardless of the control mode.

The application can explicitly turn DTR or RTS *on* or *off* with Function 46h. If the Control mode of RTS is not enable or disable, the application cannot control RTS with Function 46h because the physical device driver is controlling the signal automatically using toggling on transmit or input handshaking. If the control mode of DTR is not enable or disable, the application cannot control DTR with Function 46h because the device drive is controlling the signal automatically using input handshaking.

In CLOSE processing, when the physical device driver receives a CLOSE request packet and the COM device is still open, the physical device driver does not change the values of DTR or RTS. If the port is not open (from a Last Level Close) after processing a Close request, then, at the end of CLOSE processing, the device driver turns RTS and DTR *off* after waiting the appropriate amount of time.

Note 2

Automatic Flow Control (XON/XOFF). If bit 0 of Flags2 is set, the device driver is enabled for Automatic Transmit Flow Control. If bit 1 is set, the physical device driver is enabled for Automatic Receive Flow Control. When the physical device driver is initialized, these bits are reset, so the physical device driver is not enabled for automatic transmit or receive flow control.

An OPEN request packet does not cause the physical device driver to change the enabling or disabling state of Automatic Transmit/Receive Flow Control. If Automatic Transmit Flow Control is enabled and the COM device is not already open, an OPEN request packet causes the physical device driver to act as if it had not received an XOFF. This OPEN also causes the device driver to act as if it has not transmitted an XOFF, if Automatic Receive Flow Control is enabled.

Automatic Transmit Flow Control (XON/XOFF): When XON and XOFF flow control during transmission is enabled, the device driver stops sending data to the transmit hardware when an XOFF is received, and resumes sending data to the transmit hardware when an XON is received.

Note: Although the physical device driver can transmit XON or XOFF, there are reasons why it might not be able to transmit an XON or XOFF. For example, it might be transmitting break or invalid output handshaking on modem control signals. Also, there are reasons not related to Automatic Transmit/Receive Flow Control why the physical device driver might not be able to resume transmitting data. See ASYNC_GETCOMMSTATUS.

However, the physical device driver still continues to transmit characters as a result of the transmit immediate requests (see ASYNC_TRANSMITIMM). The physical device driver also transmits XON and XOFF because of Automatic Receive Flow Control. When the physical device driver is in this mode, it does not pass received XON and XOFF characters to the application. Instead, the physical device driver acts upon receiving those characters and discards them.

The physical device driver can transmit additional characters before it recognizes an XOFF character that is in the receive buffer of the hardware but that it has not read. The extent of this scenario is minimized, but the combined transmit/receive Advanced BIOS request block is still used on systems that support ABIOS. If the system is relatively slow in responding to interrupts compared to the current bit rate, receive buffer overruns might not occur, but the physical device driver might seem slow in responding to an XOFF character.

If Automatic Transmit Flow Control is disabled (after being currently enabled), and transmission was not occurring because an XOFF was received or at the request of ASYNC_STOPTRANSMIT, then transmission is resumed. It is the responsibility of the application to make sure that the port is closed properly so that the physical device driver cannot transmit characters after the port is reopened.

Output handshaking, using modem control signals, is one way that the physical device driver can be told to stop transmitting. See Note 3.

Automatic Receive Flow Control (XON/XOFF): When XON and XOFF flow control during receive is enabled, the device driver transmits an XOFF when its receive queue gets almost full and an XON when its receive queue is about half-full.

When the COM device driver is in Normal mode of Automatic Receive Flow Control after the XOFF is sent, it sends no characters until the amount of data in its receive queue is reduced; then it sends an XON. This is to accommodate those systems that interpret the first character received after an XOFF as an XON, regardless of what the character actually is. The physical device driver transmits characters as a result of the transmit immediate request (Function 44h).

When the COM device driver is in the Full Duplex mode of Automatic Receive Flow Control after the XOFF is sent, it continues to send characters even though the receive queue remains almost full. This mode of the physical device driver is set by using bit 5 of Flags2.

The physical device driver cannot transmit an XOFF or XON if it is transmitting a break. Also, if the physical device driver is enabled for output handshaking with certain modem control signals and those modem control signals are not *on* , it cannot automatically transmit an XON or XOFF. *Note that a deadlock can occur if the physical device driver tries to transmit an XON and cannot.* The physical device driver will continue to transmit an XOFF or XON when transmit conditions permit, assuming the receive queue conditions still warrant it.

The physical device driver does not monitor characters being transmitted by WRITE request packets to see if any of them are XON or XOFF. It also does not monitor characters transmitted immediately. For example, the device driver does not stop transmitting characters if the application causes it to explicitly transmit an XOFF.

If Automatic Receive Flow Control is enabled (after being currently disabled), the physical device driver immediately checks the receive queue level to see if an XOFF needs to be transmitted. An XON is never transmitted immediately when this function is enabled. The physical device driver automatically transmits an XON character *only* after it has automatically transmitted an XOFF character.

If Automatic Receive Flow Control is disabled (after being currently enabled) and transmission was not occurring because of the automatic transmission of an XOFF character (in the Normal mode of Automatic Receive Flow Control only), the physical device driver transmits an XON and transmission is resumed, if possible. Notice that transmission might not be taking place for other reasons (see ASYNC_GETCOMMSTATUS).

If Normal Automatic Receive Flow Control is currently enabled and transmission is not occurring because of the automatic transmission of an XOFF character, then when the physical device driver is called to enable Full-Duplex Automatic Receive Flow Control, the physical device driver immediately begins sending data regardless of the state of the receive queue. An XON character is sent only when the receive queue becomes half-full.

If the physical device driver has previously automatically transmitted an XOFF and a CLOSE request packet is received, and, after processing this Close request the port will not be open, the physical device driver automatically transmits an XON, if possible. It is the responsibility of the application to make sure that the port is closed properly so that the physical device driver cannot transmit characters after the port is reopened.

Input handshaking using modem control signals is one way that the device driver can tell another device to stop transmitting. See Note 1.

XON and XOFF Characters: The value of these bytes in the device control block determine the value of the XON and XOFF characters used for automatic transmit and receive flow control. When the XON and XOFF characters are referred to in the Category 01h IOCtls, the reference is to the value of the XON and XOFF characters as determined by this IOCtl.

When the physical device driver is first initialized, the XON character is 11h and the XOFF character is 13h. An OPEN request packet, when the COM device is not already open, causes the XON character to be set to 11h and the XOFF character to be set to 13h. If the XON and XOFF characters are set equal with this IOCtl, the results are *undefined* .

Note 3

Output Handshaking Using CTS, DSR, and DCD. Bits 3, 4, and 5 of Flags1 control Output Handshaking Using CTS, DSR, and DCD, respectively. If the bit is set, output handshaking for the appropriate modem control signal is enabled. Output Handshaking mode can be enabled for any combination of CTS, DSR, or DCD because bits 3, 4, and 5 of Flags1 can be set independently. The data is not transmitted unless all the lines enabled for output handshaking are up.

When the physical device driver is initialized, bits 3 and 4 of Flags1 are set, and bit 5 of Flags1 is reset. Therefore, initially the device driver is enabled for output handshaking using CTS and DSR, but disabled for output handshaking using DCD. Except for attachment to special devices or special cables, output handshaking using DCD should not be enabled.

Disabling Output Handshaking Using CTS or DSR causes unexpected results when the system is attached to data terminal devices or to data communications devices that toggle CTS or DSR. If the device driver is enabled for this mode of operation, it is affected in the following manner if the appropriate modem signals are *off*:

- The physical device driver is unable to move data from the physical device driver transmit queue to the transmit hardware.
- The physical device driver is unable to transmit a character immediately (Function 44h) so the character is "remembered" by the device driver.
- The physical device driver is unable to automatically transmit XONs and XOFFs. (The physical device driver might try to transmit XONs and XOFFs as a result of Automatic Receive Flow Control enabled.)
- The physical device driver generates a break immediately, if requested.
- The value of CTS, DSR, and DCD does not affect how the physical device driver controls RTS and DTR.

An OPEN request packet does not cause the physical device driver to change the value of bits 3, 4, and 5 of Flags1. It maintains the state of this mode of operation across OPEN request packets.

On devices with a transmit holding register and transmit shift register, the transmit holding register is always given another character to transmit when it empties (even though a character can still be in the transmit shift register) unless the physical device driver determines that it is no longer allowed to transmit. The physical device driver always attempts to detect a change in the modem status signals (CTS, DSR, DCD) before transmitting more data.

Note 4

Input Sensitivity Using DSR. Bit 6 of Flags1 controls input sensitivity using DSR. If the bit is set, input sensitivity using DSR is enabled. When the physical device driver is initialized, bit 6 of Flags1 is set; so initially the physical device driver is enabled for input sensitivity using DSR.

Note: Disabling input sensitivity using DSR causes unexpected results when the system is attached to data terminal devices or to data

communications devices that toggle DSR when they generate spurious data that the system should not receive.

If the physical device driver is enabled for this mode of operation, it throws away all data input from the receive hardware when DSR is *off*. If the physical device driver processes a change in the DSR modem control signal from *on* to *off* or from *off* to *on* at the same time that it inputs a character from the receive hardware, then it still accepts the last characters. This can cause the physical device driver to attempt to process invalid data for one service period of the receive hardware. Therefore, it is required that the change in the modem control signal be processed *before* the physical device driver attempts to receive data from the receive hardware (see Note 3), or that the received data be saved until a change in modem status (during the same hardware service instance) is determined.

An OPEN request packet does not cause the physical device driver to change the value of bit 6 of Flags1. The physical device driver maintains the state of this mode of operation across OPEN request packets.

Note 5

Error Replacement Character. The Flags2 bit 2 controls the enabling of error replacement character processing. If set, processing is enabled. When the physical device driver is initialized, this bit is reset, so initially the physical device driver is not enabled for the error replacement character. When the COM device is not already open, an OPEN request packet causes this bit to be reset, disabling error replacement character processing.

When the physical device driver is initialized, the error replacement character is 00h. When the COM device is not already open an OPEN request packet causes the error replacement character to be set back to 00h.

If error replacement character processing is disabled, the following applies:

- If a parity or framing error occurs and the character with the error is available in the receive hardware buffer, it is placed in the device driver receive queue.
- If a hardware or receive queue overrun occurs, nothing is placed into the receive queue to designate an overrun.

If error replacement character processing is enabled, the following applies:

- If a parity or framing error occurs, the error replacement character (if available) is placed into the physical device driver receive queue instead of the character in the receive hardware buffer.
- If a hardware buffer overrun occurs, the error replacement character is placed into the physical device driver receive queue to mark the position where the receive overrun occurred. If valid data is in the receive hardware buffer, it is placed into the device driver receive queue. The processing of the valid data takes place after the hardware buffer overrun condition is recorded in the device driver receive queue.
- If a physical device driver receive queue overrun occurs, the last character in the receive queue is replaced with the error replacement character. This allows the application to know the position where the error occurred. This error replacement, if enabled, always takes precedence over an error replacement or break replacement event that occurred for the same character.

Regardless of whether error replacement character processing is enabled, null stripping and checking for XON/XOFF characters does not occur if the character had an error. This IOCtl can be used to change the error replacement character by changing the byte representing the error replacement character.

Note 6

Null Stripping. Bit 3 in Flags2 controls the enabling of null stripping processing. If set, null stripping processing is enabled. When the physical device driver is initialized, this bit is reset, so initially the physical device driver is not enabled for null stripping. When the COM device is not already open an OPEN request packet causes this bit to be reset, disabling null stripping.

If the physical device driver is enabled for null stripping when characters are read in from the receive hardware, any *non-error* or *non-break* characters with a value of 00h are discarded, not checked (even if the XON or XOFF character has been set to 00h), and not placed into the physical device driver receive queue.

Note: Simultaneously setting the XON or XOFF character to 00h, enabling Automatic Transmit Flow Control, and enabling null stripping can cause unexpected results, but is not considered an error condition by the physical device driver error checking logic.

Note 7

Break Replacement Character. Bit 4 in Flags2 controls the enabling of break replacement character processing. If set, processing is enabled. When the physical device driver is initialized, this bit is reset, so initially the physical device driver is not enabled for the break replacement character.

When the COM device is not already open, an OPEN request packet causes this bit to be reset, disabling break replacement character processing. When the physical device driver is initialized, the break replacement character is 00h. When the COM device is not already open, an OPEN request packet causes the break replacement character to be reset back to 00h.

If break replacement character processing is disabled, the device driver does not place any character in the physical device driver receive queue when it detects a break condition on the line. A detected break condition has no effect on XON/XOFF detection. If break replacement character processing is enabled, and if the physical device driver detects a break condition, it places the break replacement character in the device driver receive queue.

If break replacement character processing is enabled, null stripping and checking for XON/XOFF characters do not operate on the break replacement character. This IOCtl can be used to change the break replacement character by changing the byte representing the break replacement character.

If a parity or framing error is generated due to the reception of a break, error replacement processing is not performed (except for the *overrun* condition); instead, break replacement processing is performed.

Note 8

Write Timeout. Bit 0 in Flags3 controls the characteristics of Write Timeout processing. If the bit is 0, Write Timeout processing uses the value in the Write Timeout WORD in the device control block. If the bit is 1, Write Timeout processing is infinite timeout.

The value in the Write Timeout WORD is in .01 second units, based on 0 (where 0 = .01 seconds). The physical device driver is considered to be doing Normal Write Timeout processing when the Write Timeout WORD is used.

During Normal Write Timeout processing, if the physical device driver does not give any data to the transmit hardware from the transmit queue within the period of time specified by the Write Timeout WORD, the request is completed. The accuracy of the timeout period can be determined by the request packet, which is blocked in the device driver, and by how long it takes for the thread to be dispatched once it is made ready by the expiration of the timeout period. The accuracy of the timeout period can also be determined by the accuracy of the device driver timer tick processing. If any data has been given to the transmit hardware in that timeout period, the device driver waits again for the specified period of time to see if any more data has been transmitted.

If the timeout period is changed by this IOCtl to Infinite Timeout, the new time can take effect immediately or it can take effect after the next character is written.

During Write Infinite Timeout processing, the request is not completed until all the data from the request has been given to the transmit hardware. The thread of the Write request does not return to the system until the request is completed. The physical device driver checks at least every minute to see if an IOCtl has changed the Write Timeout processing characteristics. This can occur almost immediately (accuracy can be determined by the request packet, which is blocked, or by device driver timer ticks), and ensures that the device driver periodically checks to see if Write Infinite Timeout processing has been changed to Normal Write Timeout processing. The Write Timeout characteristics can be changed in the middle of the processing of a Write request and the new timeout attribute is guaranteed to eventually take effect. When the physical device driver is initialized, Normal Write Timeout processing is in effect.

When the physical device driver receives an OPEN request packet for the port and the port is not already open, the value in the Write Timeout WORD is set to one minute. The current Write Timeout processing characteristics (normal or infinite) are not affected.

Note 9

Read Timeout. Bits 2, 1 of Flags3 control the Read Timeout processing characteristics of the physical device driver. The three possible types of Read Timeout processing are:

Normal	Bits 2, 1 = 0, 1
Wait-For-Something	Bits 2, 1 = 1, 0
No-Wait	Bits 2, 1 = 1, 1

The value in the Read Timeout WORD is in .01 second units, based on 0 (where 0 = .01 seconds). The physical device driver uses the value in the Read Timeout WORD for Normal and Wait-For-Something Read Timeout processing. The accuracy of the time interval can be determined by the request, which is blocked in the physical device driver, or by the device driver timer ticks.

If the physical device driver is doing Normal Read Timeout processing, the device driver waits for the amount of time specified in the Read Timeout WORD. The request is completed after that interval of time elapses if no more data has been received for the request. If any data is received by the physical device driver from the receive hardware for the request (including XON/XOFF characters), it waits the specified period of time is for more data to arrive. However, in the following two cases, the current interval of time will continue to be waited on without starting to wait from the beginning of the interval again:

- If input sensitivity using DSR is enabled and the value of the DSR modem control signal causes input data to be thrown away. See Note 4.
- If null stripping is enabled and a null character is stripped. See Note 6.

If the physical device driver is doing No-Wait Read Timeout processing, it does not wait for any data to be available in the receive queue. When the physical device driver begins to try to move data from the receive queue to the request, the request is completed. Whatever data is available in the receive queue at that time is moved to the request.

If the physical device driver is doing Wait-For-Something Read Timeout processing, the physical device driver processes the request initially as if it had No-Wait Timeout processing. If no data was available at the time the request would have completed due to No-Wait processing, the request is not completed. Instead, it waits for some data to be available before completing the request. However, the physical device driver does enter Normal Read Timeout processing for this request. Therefore, if no data is available after the Normal Timeout processing interval, then the request is completed anyway. The request never waits longer than it would have due to Normal Read Timeout processing.

The Read Timeout processing characteristics that apply to a given Read request are not determined until the physical device driver begins processing that request. At that time, a change to the Read Timeout processing characteristics of the physical device driver between Wait-For-Something and Normal Timeout processing might take effect for the current Read request being processed. If the timeout period is changed by this IOCtl, the new timeout period might take effect immediately or it might take effect after the next character is received from the receive hardware. When the physical device driver is initialized, Normal Read Timeout processing is in effect.

When the physical device driver receives an OPEN request packet for the port and the port is not already open, the value in the Read Timeout WORD is set to one minute and Normal Read Timeout processing characteristics are put into effect.

Note 10

Extended Hardware Buffering. This refers to the capability of the COM port's serial controller device to buffer up to 16 characters in its internal hardware Receive and Transmit buffers. This buffering capability allows the device to relieve the operating system of the high overhead associated with servicing per-character receive and transmit hardware interrupts.

On COM ports with a serial controller device that does not support Extended Hardware Buffering, bits 3 and 4 of DCB Flags3 are always set to 0, indicating that the device does not support Extended Hardware Buffering. This value is always valid and ignored as input to Function 53h. If the device does not support Extended Hardware Buffering and the application attempts to set any of the Flags3 bits 3-7, this IOCtl fails and a general failure error results.

Applications must first call ASYNC_GETDCBINFO to determine whether the device supports Extended Hardware Buffering *before* calling Function 53h.

When in conventional Programmed Input/Output (PIO) mode or when the Enhanced mode is disabled through ASYNC_SETENHANCEDMODEPARMS, the physical ASYNC device driver defaults the setting of the Extended Hardware Buffering parameter to Automatic Protocol Override. This system default can be changed by manipulating bits 3 and 4 in the Flags3 parameter of Function 53h. An application or subsystem can manually control this feature by setting Extended Hardware Buffering enabled and by manipulating the Receive Trigger Level and Transmit Buffer Load Count parameters. An application or subsystem can also set the serial device to run in Character mode by setting Extended Hardware Buffering disabled. The three settings for this feature are described below.

Automatic Protocol Override (System Default): In any system configuration where a COM port's serial controller correctly supports Extended Hardware Buffering, the physical device driver initializes that COM port to enable Automatic Protocol Override mode. This mode of the physical ASYNC device driver is defined with respect to the following device driver protocols:

- Output Handshaking Using CTS, DSR, DCD
- Automatic Transmit Flow Control
- Input Sensitivity Using DSR

When any one or more of the above protocols are enabled, the Automatic Protocol Override feature causes the Extended Hardware Buffering not to fully exploit the maximum potential performance benefit of the serial controller. Depending on which protocols are enabled, the physical device driver automatically adjusts either, or both, the Receive Trigger Level and Transmit Buffer Load Count. The following descriptions identify the changes that each of the above protocols causes when Automatic Protocol Override mode is active:

- Output Handshaking Using CTS, DSR (Default *on*)

- Output Handshaking Using DCD (Default *off*). When either of these handshaking protocols is enabled, the physical ASYNC device driver, under Automatic Protocol Override, sets the Transmit Buffer Load Count to 1. This means that it services transmit interrupts one character at a time. When these protocols are disabled, the device driver fully exploits the Extended Hardware Buffering capabilities of the serial controller by transmitting 16 characters per interrupt. The Receive Trigger Level is not affected by these protocols in Automatic Protocol Override mode.

- Automatic Transmit Flow Control (Default *off*). When this protocol is enabled, the physical ASYNC device driver, under Automatic Protocol Override, sets the Receive Trigger Level and the Transmit Buffer Load Count to 1. This means that it services both receive and transmit interrupts one character at a time. When these protocols are disabled, the physical device driver fully exploits the Extended Hardware Buffering capabilities of the serial controller by transmitting 16 characters per interrupt and setting the Receive Trigger Level to *8*. (The physical device driver is given at least 8 characters on each Receive Data Available interrupt.)

- Input Sensitivity Using DSR (Default *on*). When this protocol is enabled, the Automatic Protocol Override feature of the ASYNC device driver sets the Receive Trigger Level to 1, by default, on the respective COM port, forcing the physical device driver to service all characters received one character at a time. The Transmit Buffer Load Count is not affected by this protocol.

With all of the above listed protocols, Automatic Protocol Override allows the serial controller to remain in its FIFO-mode setting, although it is not fully exploiting the potential performance benefit from the Extended Hardware Buffering capability. The extended Receive Hardware Buffering capability remains active so that receive hardware overrun errors are much less likely to occur.

When Automatic Protocol Override mode is enabled, setting the DCB Flags3 bits for manipulating Receive Trigger Level and Transmit Buffer Load Count (bits 5, 6, and 7) has no effect. Automatic Protocol Override fully overrides any manual settings the application or subsystem might attempt to set.

Note: Having any of the above protocols enabled can significantly alter system performance characteristics with respect to activity on an asynchronous communications port. This is particularly true where the COM port is running at a high bit rate (2400 bps or higher) or where multiple COM ports are performing asynchronous I/O. This is a factor to consider when configuring a system to perform multiple serial port I/O or when developing an application that requires high speed asynchronous communications. Disabling the above protocols or setting the Device Control Block parameters to Extended Hardware Buffering enabled allows the physical device driver to fully exploit the serial controller's capability to its maximum benefit.

Extended Hardware Buffering Enabled

Setting this mode cancels the effects of Automatic Protocol Override and enables the user to fully use the Extended Hardware Buffering capabilities of the physical device driver. The Receive Trigger Level should be set to 8. (the physical device driver receives 8 characters per

interrupt) and the Transmit Buffer Load Count should be set to 16 (the physical device driver transmits 16 characters per interrupt).

Setting this mode in conjunction with any of the following device driver protocols can alter the behavior of that protocol in a significant manner:

- Output Handshaking using CTS, DSR, DCD
- Automatic Transmit Flow Control
- Input Sensitivity using DSR

For information on these protocols, refer to the description in the *OS/2 Input/Output Device Driver Reference* .

Extended Hardware Buffering Disabled

Setting this mode completely disables the Extended Hardware Buffering capabilities of the serial port controller. This mode places the serial port device into Character mode (that is, the physical device driver services both transmit and receive interrupts one character at a time). This effectively eliminates any special considerations that might have been necessary when performing asynchronous communications with Extended Hardware Buffering enabled or with the Automatic Protocol Override mode active.

On any COM port that is serviced by a serial controller that does not support FIFO mode, the physical device driver is always set to Extended Hardware Buffering disabled. Attempts to set Extended Hardware Buffering enabled or to enable the Automatic Protocol Override mode are ignored (no error returned). Whenever ASYNC_GETDCBINFO is called, the bits in DCB Flags3 identify the true state of Extended Hardware Buffering on these devices as disable.

Note: Setting the physical device driver to run with Extended Hardware Buffering disabled can significantly alter the system performance characteristics, particularly with respect to asynchronous communications I/O throughput. This is especially true where the COM port is running at a high bit rate (2400 bps or higher) or where multiple COM ports are performing asynchronous I/O.

Note 11

Receive Trigger Level. This represents the number of characters that must be received by a serial port controller that supports Extended Hardware Buffering before that device generates a receive hardware interrupt. For example, with Extended Hardware Buffering enabled, assume the Receive Trigger Level is set to 8 characters. This means that a receive hardware interrupt occurs once for every 8 characters received at that COM port. On COM ports with a serial controller device that does not support Extended Hardware Buffering, bits 5 and 6 of DCB Flags3 are always set to 0, indicating that the device is in Character mode (Receive Trigger Level=1).

The default state for bits 5 and 6 of DCB Flags3 is zero, indicating that the Automatic Protocol Override mode is active and that other device driver protocols are enabled, which forces Automatic Protocol Override to keep Receive Trigger Level set to 1.

When the physical device driver is operating in Automatic Protocol Override mode, setting the Receive Trigger Level has no effect. This parameter setting is completely ignored unless DCB Flags3 bits 3 and 4 are set to Extended Hardware Buffering enabled.

Note: System performance and asynchronous communications I/O throughput can be significantly degraded by setting Receive Trigger Level to 1 on COM ports whose serial controller device is capable of supporting Extended Hardware Buffering. This is particularly true where a COM port is sending and receiving data at high bit rates (over 2400 bps) or where multiple COM ports are simultaneously engaged in asynchronous communications.

Setting the Receive Trigger Level to 14 characters can increase the probability of getting receive hardware overrun errors. This is most likely where a COM port is sending and receiving data at high bit rates (over 2400 bps) or where multiple COM ports are simultaneously engaged in asynchronous communications.

ASYNC_GETDCBINFO always gives the actual current setting of the Receive Trigger Level, regardless of the setting of the Extended Hardware Buffering mode. In Enhanced mode, the Receive Trigger Level setting is ignored and the ASYNC device drive automatically sets the value for maximum operational efficiency. In this case, the Receive Trigger Level chosen by the physical ASYNC device driver is not reflected in Function 73h. The value that the user has specified for the Receive Trigger Level in this IOCtl is returned, even though it is ignored.

Note 12

Transmit Buffer Load Count refers to the number of characters that the physical ASYNC device driver gives to a COM port's serial controller transmit hardware on the occurrence of a transmit hardware interrupt. On COM ports with a serial controller device that does not support Extended Hardware Buffering, bit 7 of DCB Flags3 is always set to 0, indicating that the device is in Character mode (Transmit Buffer Load Count=1).

Normally, when the serial controller supports Extended Hardware Buffering and the physical device driver protocols are set to fully use this hardware capability, the Transmit Buffer Load Count is set to 16 characters. This means that every time the physical device driver gets control of the serial controller on the occurrence of a transmit hardware interrupt, 16 characters are placed in the serial controller's transmit buffer.

There might be situations where an application or communications subsystem needs to control the flow of data manually during a communications session. Setting the Transmit Buffer Load count to 1 character ensures that the physical device driver has control of the serial controller every time a character is transmitted.

When the physical device driver is operating in Automatic Protocol Override mode, setting the Transmit Buffer Load Count has no effect. This parameter setting is completely ignored unless DCB Flags3 bits 3 and 4 are set to Extended Hardware Buffering enabled.

Note: System performance and asynchronous communications I/O throughput can be significantly degraded by setting Transmit Buffer Load

Count to 1 on COM ports whose serial controller device is capable of supporting Extended Hardware Buffering. This is particularly true where a COM port is sending and receiving data at high bit rates (over 2400 bps) or where multiple COM ports are simultaneously engaged in asynchronous communications.

ASYNC_SETDCBINFO (53h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETDCBINFO (53h)
Description:
Set Device Control Block (DCB) Parameters

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_SETENHANCEDMODEPARMS (54h) - Set Enhanced Moc

ASYNC_SETENHANCEDMODEPARMS (54h) - Description

This function sets the enhanced mode parameters. This IOCtl is not currently supported by COM.SYS.

ASYNC_SETENHANCEDMODEPARMS (54h) - Enhanced Flags1

Enhanced Flags1
Has the following settings (see Note 1 for more information):

Bit 0	Enhanced mode supported by hardware. (Query only for Function 74h.)
Bit 1	Enable the enhanced mode (default)
Bits 2-3	DMA Receive Operation request. Has the following:

Bit 3	Bit 2	Description
0	0	Disable DMA Receive Capability
0	1	Enable DMA Receive Capability (default)
1	0	Dedicate a DMA channel to Receive operation
1	1	Reserved

Bits 4-5 DMA Transmit operation request. Has the following:

Bit 5	Bit 4	Description
0	0	Disable DMA Transmit Capability
0	1	Enable DMA Transmit Capability (default)
1	0	Dedicate a DMA channel to Transmit operation

	1	1	Reserved
Bit 6	Receive operation in DMA mode. (Query only for Function 74h.)		
Bit 7	Transmit operation in DMA mode. (Query only for Function 74h.)		

ASYNC_SETENHANCEDMODEPARMS (54h) - Parameter Packet

Field	Length	C Datatype
Enhanced Flags1	BYTE	UCHAR
Reserved	DWORD	ULONG

ASYNC_SETENHANCEDMODEPARMS (54h) - Data Packet Forma

None. Packet pointer must be NULL.

ASYNC_SETENHANCEDMODEPARMS (54h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and none of the enhanced parameter settings for this COM device is changed.

ASYNC_SETENHANCEDMODEPARMS (54h) - Remarks

This function is used to disable/enable the Enhanced mode operations or to control the Receive/Transmit operations on a COM port with the Enhanced UART or compatibles. As a default, the physical device driver automatically runs I/O operations in Enhanced mode, which is either DMA mode or Enhanced FIFO mode. In DMA mode, the data transfer from port to memory, or from memory to port, can be done directly by the DMA chip, alleviating the CPU operations and gaining maximum performance gain. A DMA-mode operation requires a system resource called the *DMA channel*.

In Enhanced FIFO mode, the full capacity of Extended Hardware Buffering is used automatically by the physical device driver. In both operations, the ABIOS function interfaces are used by the physical device driver to maintain full compatibility with existing communication protocols. The Extended Hardware Buffering Flags3 bit settings of ASYNC_SETDCBINFO are ignored.

The user can turn the Enhanced mode *off* with this function, if desired. In this case, a COM port behaves as if it is a conventional serial device with Extended Hardware Buffering capability. The Extended Hardware Buffering bit settings of Flags3 (Function 53h) are used to control the hardware FIFO buffer.

If a general failure error is not returned, then the actions described below are taken by the physical device driver. If the Data Packet pointer is not NULL, the IOCtl fails with the ERROR_GEN_FAILURE return code.

To maintain hardware compatibility, the application must call ASYNC_GETENHANCEDMODEPARMS, before the Function 54h is used. This allows the Reserved bits to be set correctly in a future release of the device driver. By calling Function 74h first, the application can maintain the state of the physical device driver for a mode that the application is not aware of.

Note 1

Enhanced Flags1. Bits 0, 6, and 7 are only for querying the status of the Enhanced mode (Function 74h) and are ignored when the user calls Function 54h. Before setting bits 1-5, the user must call Function 74h to ensure that a COM port supports the Enhanced mode and must check to see if bit 0 of the Enhanced Flags1 is set. If bit 1 is set in Function 54h, the physical device driver generates the general failure error return code.

As a default, when the Enhanced mode is supported, it is enabled and the Enable Enhanced Mode bit is set. The user can disable the Enhanced mode by resetting this bit. When Enhanced mode is disabled, the settings of bits 2-5 are ignored and the enhanced FIFO/DMA capabilities of the hardware are not exploited. It is important that the user not disable the Enhanced mode unless it is required to control the hardware FIFO buffer manually. In Enhanced mode, the physical device driver automatically controls the hardware FIFO buffer for its maximum efficiency. The advanced function interfaces provided by the hardware allow a full compatibility with the existing RS232-C communication protocols.

When the Disable DMA Receive Capability option is chosen, the device driver does not try to use the DMA capability for receive operations. Instead, the physical device driver runs receive operations in Enhanced FIFO mode where the full FIFO capability and the enhanced ABIOS function interfaces are automatically exploited. The efficiency of the Enhanced FIFO-mode operations is not as good as the DMA-mode operation, but is better than the Character-mode operation. This option allows the user to control the use of a limited number of the available DMA channels in the system. When the Enhanced mode is disabled, the system runs in conventional mode and the advanced function features provided by the hardware are not utilized.

The initial default system uses the Enable DMA Receive Capability option. In Enhanced mode, the physical device driver tries to use a DMA channel at Open request time. If there is no DMA channel available at that moment, the operation defaults to Enhanced FIFO-mode operation. If the physical device driver can successfully allocate a DMA channel, then the receive operation operates in DMA mode. When the Enhanced mode is disabled, no attempt is made to allocate a DMA channel.

When the Dedicate a DMA Channel to Receive Operation option is requested, the physical device driver tries to allocate a DMA channel as a dedicated one for receive operations. If a DMA channel cannot be allocated at the time of request, the physical device driver returns the general failure error. If a DMA channel can be allocated successfully by Function 54h, it is not deallocated until the user issues Function 54h again with another option such as Enable DMA Receive Operation, Disable DMA Receive Operation, or Disable Enhanced Mode.

Note: If the user requests to switch the state of DMA Receive Capability while a receive operation is in process, there is a chance of loss of data. It is important that the user check the emptiness of the device driver receive and transmit queues through ASYNC_GETINQUECOUNT, and ASYNC_GETOUTQUECOUNT. Before requesting the DMA-mode switch, the user must stop transmitting and communication protocols must be employed to ensure the transmitting system on the other end of the line has stopped transmitting.

When the Disable DMA Transmit Capability option is chosen, the physical device driver does not try to use the DMA capability for transmit operations. Instead, the physical device driver runs transmit operations in Enhanced FIFO-mode, where the full FIFO capability and the enhanced ABIOS function interfaces are automatically exploited. The efficiency of the Enhanced FIFO-mode operations is not as good as the DMA-mode operation, but is better than the Character-mode operation. This option allows the user to control the use of a limited number of the available DMA channels in the system. When the Enhanced mode is disabled, the system runs in conventional mode and the advanced function features provided by the hardware are not utilized.

The initial default system uses the Enable DMA Transmit Capability option. In Enhanced mode, with this option, the physical device driver tries to use a DMA channel at each transmit request, and if there is no DMA channel available at that moment, then the operation defaults to Enhanced FIFO-mode operation. If the physical device driver can successfully allocate a DMA channel, then the transmit operation operates in DMA mode. When the Enhanced mode is disabled, no attempt is made to allocate a DMA channel.

When the Dedicate a DMA Channel to Transmit Operation option is requested, the physical device driver tries to allocate a DMA channel as a dedicated one for transmit operation. If a DMA channel cannot be allocated at the time of request, the physical device driver returns the general failure error. If a DMA channel can be allocated successfully by Function 54h, it is not deallocated until the user issues Function 54h again with another option such as Enable DMA Transmit Operation, Disable DMA Transmit Operation, or Disable Enhanced Mode.

Note: If the user requests to switch the state of DMA Transmit Capability while a transmit operation is in process, there is a chance of loss of data. It is important that the user check the emptiness of the device driver transmit queue through ASYNC_GETOUTQUECOUNT.

The First Level Open or Last Level Close does not affect the bit settings of Enhanced Flags1.

ASYNC_SETENHANCEDMODEPARMS (54h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_SETENHANCEDMODEPARMS (54h)
Description:
Set Enhanced Mode Parameters

Select an item:
Description
Parameter Packet Format

Data Packet Format
Returns
Remarks

ASYNC_GETBAUDRATE (61h) - Query Current Bit Rate

ASYNC_GETBAUDRATE (61h) - Description

This function returns the bit rate.

ASYNC_GETBAUDRATE (61h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETBAUDRATE (61h) - Bit Rate

Bit Rate
A binary integer representing the actual bit rate of the COM device in bits-per-second, rounded to the nearest whole number.

ASYNC_GETBAUDRATE (61h) - Data Packet Format

Field	Length	C Datatype
Bit Rate	WORD	USHORT

ASYNC_GETBAUDRATE (61h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETBAUDRATE (61h) - Remarks

If a general failure error is not returned, the physical device driver returns the current bit rate of the COM device.

Note: If this function is called when the current bit rate setting of a COM port is greater than what can be stored in a *1-WORD* field, the device driver sets the bit rate to 1200 bps (default value) and returns 1200 bps to the user.

ASYNC_GETBAUDRATE (61h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETBAUDRATE (61h)
Description:
Query Current Bit Rate

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETLINECTRL (62h) - Query Line Characteristics

ASYNC_GETLINECTRL (62h) - Description

This function returns the line characteristics (data bits, parity, stop bits, break).

ASYNC_GETLINECTRL (62h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETLINECTRL (62h) - Data Bits

Data Bits
See ASYNC_SETLINECTRL.

ASYNC_GETLINECTRL (62h) - Parity

Parity

See ASYNC_SETLINECTRL.

ASYNC_GETLINECTRL (62h) - Stop Bits

Stop Bits

See ASYNC_SETLINECTRL.

ASYNC_GETLINECTRL (62h) - Transmitting Break

Transmitting Break

0	not currently transmitting break
1	currently transmitting break

ASYNC_GETLINECTRL (62h) - Data Packet Format

Field	Length	C Datatype
Data Bits	BYTE	BYTE
Parity	BYTE	BYTE
Stop Bits	BYTE	BYTE
Transmitting Break	BYTE	BYTE

Related C Data Structure

The LINECONTROL data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

ASYNC_GETLINECTRL (62h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETLINECTRL (62h) - Remarks

If a general failure error is not returned, the physical device driver returns the line characteristics as defined.

ASYNC_GETLINECTRL (62h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETLINECTRL (62h)
Description:
Query Line Characteristics

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_EXTGETBAUDRATE (63h) - Extended Query Bit Rate

ASYNC_EXTGETBAUDRATE (63h) - Description

This function returns the extended query bit rate in doublewords for a bit rate higher than 19200 bps.

ASYNC_EXTGETBAUDRATE (63h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_EXTGETBAUDRATE (63h) - Current Bit Rate

Current Bit Rate
The binary integer representing the actual bit rate in bits-per-seconds set for a COM port.

ASYNC_EXTGETBAUDRATE (63h) - Fraction of Current Bit Rate

Fraction of Current Bit Rate

The binary integer representing the fraction of the actual current bit rate set for a COM port.

ASYNC_EXTGETBAUDRATE (63h) - Minimum Bit Rate Supported

Minimum Bit Rate Supported
The binary integer representing the minimum bit rate (in bits-per-second) port for a COM port.

ASYNC_EXTGETBAUDRATE (63h) - Fraction of Minimum Bit Rate

Fraction of Minimum Bit Rate Supported
The binary integer representing the fraction of the minimum bit rate supported for a COM port.

ASYNC_EXTGETBAUDRATE (63h) - Maximum Bit Rate Supported

Maximum Bit Rate Supported
The binary integer representing the maximum bit rate (in bits-per-second) supported for a COM port. Depending on overall system overhead and the electrical characteristics of the hardware cables and serial device adapter type, the actual value of the maximum bit rate supported might be lower than this.

ASYNC_EXTGETBAUDRATE (63h) - Fraction of Maximum Bit Rate

Fraction of Maximum Bit Rate Supported
The binary integer representing the fraction of the maximum bit rate supported for a COM port.

ASYNC_EXTGETBAUDRATE (63h) - Data Packet Format

Field	Length	C Datatype
Current Bit Rate	DWORD	ULONG
Fraction of Current Bit Rate	BYTE	UCHAR
Minimum Bit Rate Supported	DWORD	ULONG
Fraction of Minimum Bit Rate Supported	BYTE	UCHAR
Maximum Bit Rate Supported	DWORD	ULONG

Fraction of Maximum Bit Rate Supported	BYTE	UCHAR

ASYNC_EXTGETBAUDRATE (63h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_EXTGETBAUDRATE (63h) - Remarks

If a general failure error is not returned, the physical device driver returns the bit rate values as defined. This function is extended from the current "Function 61h - Query Bit Rate".

ASYNC_EXTGETBAUDRATE (63h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_EXTGETBAUDRATE (63h)
Description:
Extended Query Bit Rate

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETCOMMSTATUS (64h) - Query COM Status

ASYNC_GETCOMMSTATUS (64h) - Description

This function returns the COM status.

ASYNC_GETCOMMSTATUS (64h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETCOMMSTATUS (64h) - COMM Status Byte

COMM Status Byte

If equal to 1, the condition is TRUE. If equal to 0, the condition is FALSE.

Bit 0	Transmit status (Tx) waiting for CTS to be turned *on*. See Note 3 of ASYNC_SETDCBINFO.
Bit 1	Tx waiting for DSR to be turned *on*. See Note 3 of Function 53h.
Bit 2	Tx waiting for DCD to be turned *on*. See Note 3 of Function 53h.
Bit 3	Tx waiting because XOFF received. Behaves as if XOFF received (Function 47h). See Note 2 of Function 53h. Characters are transmitted immediately (Function 44h). When the device driver is in this state, it automatically transmits XONs and XOFFs due to Automatic Receive Flow Control.
Bit 4	Tx waiting because XOFF transmitted. See Note 2 of Function 53h. Characters are still transmitted immediately. When the physical device driver is in this state, it automatically transmits XONs due to Automatic Receive Flow Control.
Bit 5	Tx waiting because break being transmitted. See ASYNC_SETBREAKON.
Bit 6	Character waiting to transmit immediately. See ASYNC_TRANSMITIMM.
Bit 7	Receive waiting for DSR to be turned *on*. See Note 4 of Function 53h.

Transmit status (Tx) indicates why transmission might not occur, regardless of whether there is data to transmit. However, in order for the status to reflect that the physical device driver is waiting for a given condition, the device driver must be enabled for that condition.

For example, 00000001 indicates that the physical device driver has put receive characters in the physical device driver receive queue and is not waiting to transmit a character immediately (Function 44h). Characters are not transmitted from the physical device driver transmit queue when using CTS for output handshaking because CTS does not have the proper value.

ASYNC_GETCOMMSTATUS (64h) - Data Packet Format

Field	Length	C Datatype
COMM Status Byte	BYTE	UCHAR

ASYNC_GETCOMMSTATUS (64h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETCOMMSTATUS (64h) - Remarks

If a general failure error is not returned, the physical device driver returns the COM device current status.

ASYNC_GETCOMMSTATUS (64h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETCOMMSTATUS (64h)
Description:
Query COM Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETLINESTATUS (65h) - Query Transmit Data Status

ASYNC_GETLINESTATUS (65h) - Description

Returns the transmit data status.

ASYNC_GETLINESTATUS (65h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETLINESTATUS (65h) - Transmit Status

Transmit Status
Returned as bit-significant values. If the bit is 1, the condition is TRUE. If the bit is 0, the condition is FALSE. The number at the beginning of the description is the bit position number. The bit positions go from least to most significant.

Bit 0	WRITE request packets in progress or queued
Bit 1	Data in the physical device driver transmit queue
Bit 2	Transmit hardware is currently transmitting data
Bit 3	Character waiting to be transmitted immediately
Bit 4	Waiting to automatically transmit an XON
Bit 5	Waiting to automatically transmit an XOFF
Bit 6	Undefined
Bit 7	Undefined

ASYNC_GETLINESTATUS (65h) - Data Packet Format

Field	Length	C Datatype
Transmit Status	BYTE	UCHAR

ASYNC_GETLINESTATUS (65h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETLINESTATUS (65h) - Remarks

If a general failure error is not returned, the physical device driver returns the current transmit status of the COM device.

ASYNC_GETLINESTATUS (65h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETLINESTATUS (65h)
Description:
Query Transmit Data Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETMODEMOUTPUT (66h) - Query Modem Output Signa

ASYNC_GETMODEMOUTPUT (66h) - Description

Returns the modem control output signals.

ASYNC_GETMODEMOUTPUT (66h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETMODEMOUTPUT (66h) - Modem Control Output Statu

Modem Control Output Status

If a bit has a value of 1, the condition is *on*. If a bit has a value of 0, the condition is *off*.

Bit 0	Data Terminal Ready (DTR)
Bit 1	Request To Send (RTS)
Bits 2-7	Undefined

ASYNC_GETMODEMOUTPUT (66h) - Data Packet Format

Field	Length	C Datatype
Modem Control Output Status	BYTE	UCHAR

ASYNC_GETMODEMOUTPUT (66h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETMODEMOUTPUT (66h) - Remarks

If a general failure error is not returned, the physical device driver returns the current modem control output signals of the COM device.

ASYNC_GETMODEMOUTPUT (66h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETMODEMOUTPUT (66h)
Description:
Query Modem Output Signals

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETMODEMINPUT (67h) - Query Current Modem Input Si

ASYNC_GETMODEMINPUT (67h) - Description

Returns the modem control input signals.

ASYNC_GETMODEMINPUT (67h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETMODEMINPUT (67h) - Modem Control Input Status

Modem Control Input Status

If a bit has a value of 1, the condition is *on*. If a bit has a value of 0, the condition is *off*.

Bit 0-3	Undefined
Bit 4	Clear To Send (CTS)
Bit 5	Data Set Ready (DSR)
Bit 6	Ring Indicator (RI)
Bit 7	Data Carrier Detect (DCD)

ASYNC_GETMODEMINPUT (67h) - Data Packet Format

Field	Length	C Datatype
Modem Control Input Status	BYTE	UCHAR

ASYNC_GETMODEMINPUT (67h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETMODEMINPUT (67h) - Remarks

If a general failure error is not returned, the physical device driver returns the current modem control input signals of the COM device.

ASYNC_GETMODEMINPUT (67h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETMODEMINPUT (67h)
Description:
Query Current Modem Input Signals

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETINQUECOUNT (68h) - Query Number of Characters ir

ASYNC_GETINQUECOUNT (68h) - Description

Returns the number of characters in the receive queue.

ASYNC_GETINQUECOUNT (68h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETINQUECOUNT (68h) - Number of Characters Queued

Number of Characters Queued
Binary integer with the number of received characters in the device driver receive queue. This is a memory buffer between the memory pointed to by the READ request packet and the receive hardware for this COM device. The application cannot assume

that there are no unsatisfied Read requests if there are characters in the device driver receive queue.

Note: The behavior of data movement between the Read request and the receive queue can change with each release of the physical device driver. Applications should not have a dependency on this information.

ASYNC_GETINQUECOUNT (68h) - Size of Receive Queue

Size of Receive Queue

Binary integer with the size of the physical device driver receive queue. Applications should be independent of the receive queue being sized (fixed or not fixed). The information in this field allows the application to get the size of the receive queue. The current size of the receive queue is approximately 1KB, but is subject to change.

Using this information, the application should avoid device driver receive queue overruns by using an application-to-application block protocol with the system communicating with the application.

ASYNC_GETINQUECOUNT (68h) - Data Packet Format

Field	Length	C Datatype
Number of Characters Queued	WORD	USHORT
Size of Receive Queue	WORD	USHORT

Related C Data Structure

The RXQUEUE data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

ASYNC_GETINQUECOUNT (68h) - Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported, and valid information is not returned in the Data Packet.

ASYNC_GETINQUECOUNT (68h) - Remarks

If a general failure error is not returned, the physical device driver returns the information.

ASYNC_GETINQUECOUNT (68h) -

Category:
IOCTL_ASYNC (01h)
Function:

ASYNC_GETINQUECOUNT (68h)

Description:

Query Number of Characters in Receive Queue

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETOUTQUECOUNT (69h) - Query Number of Characters

ASYNC_GETOUTQUECOUNT (69h) - Description

Returns the number of characters in the transmit queue.

ASYNC_GETOUTQUECOUNT (69h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETOUTQUECOUNT (69h) - Number of Characters Queu

Number of Characters Queued

Binary integer with the number of characters ready to be transmitted in the physical device driver transmit queue. This is a memory buffer between the memory pointed to by the WRITE request packet and the transmit hardware for this COM device. If the transmit queue is empty, the application cannot assume that all Write requests have completed or that no Write requests are outstanding.

Note: The behavior of data movement between the Write request and the transmit queue can change with each release of the physical device driver. Applications should not be dependent on this information.

ASYNC_GETOUTQUECOUNT (69h) - Size of Transmit Queue

Size of Transmit Queue

Binary integer with the size of the physical device driver transmit queue. Applications should be independent of the transmit queue size (fixed or not fixed). The information in this field allows the application to get the size of the transmit queue, but is subject to change.

ASYNC_GETOUTQUECOUNT (69h) - Data Packet Format

Field	Length	C Datatype
Number of Characters Queued	WORD	USHORT
Size of Transmit Queue	WORD	USHORT

Related C Data Structure

The RXQUEUE data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

ASYNC_GETOUTQUECOUNT (69h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETOUTQUECOUNT (69h) - Remarks

If a general failure error is not returned, the physical device driver returns the information.

ASYNC_GETOUTQUECOUNT (69h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETOUTQUECOUNT (69h)
Description:
Query Number of Characters in Transmit Queue

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETCOMMERROR (6Dh) - Query COM Error

ASYNC_GETCOMMERROR (6Dh) - Description

Returns COM Error (retrieves and clears the COM device error information).

ASYNC_GETCOMMERROR (6Dh) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETCOMMERROR (6Dh) - COM Error WORD (COMERR)

COM Error WORD (COMERR)

The appropriate bits in COM Error WORD are set by the physical device driver when the events described below occur. COM Error WORD is not cleared unless this function is performed by the physical device driver or an OPEN request packet is received by the physical device driver and the COM device is not already open (First Level Open). See Note 5 of ASYNC_SETDCBINFO.

Bit 0	Receive queue overrun. No room in the physical device driver receive queue to put a character read in from the receive hardware.
Bit 1	Receive hardware overrun. A character was not read from the hardware before the next character arrived, causing a character to be lost.
Bit 2	The hardware detected a parity error.
Bit 3	The hardware detected a framing error.
Bits 4-15	Undefined.

ASYNC_GETCOMMERROR (6Dh) - Data Packet Format

Field	Length	C Datatype
COM Error WORD (COMERR)	WORD	USHORT

ASYNC_GETCOMMERROR (6Dh) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, valid information is not returned in the Data Packet, and the COM Error WORD is not cleared.

ASYNC_GETCOMMERROR (6Dh) - Remarks

If a general failure error is not returned, the physical device driver returns the current value of the error WORD and then clears it.

ASYNC_GETCOMMERROR (6Dh) -

Category:

IOCTL_ASYNC (01h)
Function:
ASYNC_GETCOMMERROR (6Dh)
Description:
Query COM Error

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETCOMMEVENT (72h) - Query COM Event Information

ASYNC_GETCOMMEVENT (72h) - Description

This function returns COM event information (retrieves and clears the COM device event WORD).

ASYNC_GETCOMMEVENT (72h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETCOMMEVENT (72h) - COM Event WORD

COM Event WORD

The appropriate bits in the COM Event WORD are set by the device driver when the events described below occur. The COM Event WORD is not cleared unless this function is performed by the physical device driver or an OPEN request packet is received by the physical device driver and the COM device is not already open.

Bit 0	Set when any character is read from the COM device receive hardware and placed in the receive queue.
Bit 1	Set whenever the serial port controller generates a Receive Timeout Interrupt during a Receive request. This bit is always zero when the serial port controller does not support Extended Hardware Buffering.
Bit 2	Set when the last character in the physical device driver transmit queue is sent to the COM device transmit hardware. Data in any outstanding Write requests still might need to be sent.
Bit 3	Set when the Clear To Send (CTS) signal changes state.
Bit 4	Set when the Data Set Ready (DSR) signal changes state.
Bit 5	Set when the Data Carrier Detect (DCD) signal changes state.
Bit 6	Set when a break is detected.
Bit 7	Set when a parity, framing, or receive hardware (or receive queue) overrun error occurs.

Bit 8	Set when trailing edge of Ring Indicator is detected.
Bits 9-15	Undefined.

ASYNC_GETCOMMEVENT (72h) - Data Packet Format

Field	Length	C Datatype
COM Event WORD	WORD	USHORT

ASYNC_GETCOMMEVENT (72h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, valid information is not returned in the Data Packet, and the event WORD is not cleared.

ASYNC_GETCOMMEVENT (72h) - Remarks

If a general failure error is not returned, the physical device driver returns the current value of the event WORD and then clears it.

ASYNC_GETCOMMEVENT (72h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETCOMMEVENT (72h)
Description:
Query COM Event Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETDCBINFO (73h) - Query Device Control Block (DCB) F

ASYNC_GETDCBINFO (73h) - Description

This function returns Device Control Block (DCB) information.

ASYNC_GETDCBINFO (73h) - Parameter Packet Format

None. Packet pointer must be NULL.

ASYNC_GETDCBINFO (73h) - Write Timeout

Write Timeout

Specifies the time period used for Write Timeout processing. See Note 8 of ASYNC_SETDCBINFO. The value is in .01 second units based on zero (where 0 = .01 seconds).

ASYNC_GETDCBINFO (73h) - Read Timeout

Read Timeout

Specifies the time period used for Read Timeout processing. See Note 9 of Function 53h. The value is in .01 second units based on zero (where 0 =.01 seconds).

ASYNC_GETDCBINFO (73h) - Flags1

Flags1

Has the following bits:

Bits 0-1 — DTR Control mode. Has the following:

Bit 1	Bit 0	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Invalid input. Results in a *general failure* error.

Bit 2 — Reserved (returned as 0)

Bit 3 — Enable output handshaking using CTS

Bit 4 — Enable output handshaking using DSR

Bit 5 — Enable output handshaking using DCD

Bit 6 — Enable input sensitivity using DSR

Bit 7 — Reserved (returned as 0)

ASYNC_GETDCBINFO (73h) - Flags2

Flags2

Has the following bits:

Bit 0	Enable Automatic Transmit Flow Control (XON/XOFF)
Bit 1	Enable Automatic Receive Flow Control (XON/XOFF)
Bit 2	Enable error replacement character
Bit 3	Enable null stripping (remove null bytes)
Bit 4	Enable break replacement character
Bit 5	Automatic Receive Flow Control

0 = Normal
1 = Full-Duplex

Bits 6-7 RTS Control mode. Has the following:

Bit 7	Bit 6	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Toggling on transmit

ASYNC_GETDCBINFO (73h) - Flags3

Flags3

Has the following bits:

Bit 0	Enable Write Infinite Timeout processing

Bits 1-2 Read Timeout processing. Has the following:

Bit 2	Bit 1	Description
0	1	Normal Read Timeout processing
1	0	Wait-For-Something, Read Timeout processing
1	1	No-Wait, Read Timeout processing

Bits 3-4 Extended Hardware Buffering. Has the following:

Bit 4	Bit 3	Description
0	0	Not supported
0	1	Extended Hardware Buffering Disabled
1	0	Extended Hardware Buffering Enabled
1	1	Automatic Protocol Override

Bits 5-6 Receive Trigger Level. Has the following:

Bit 6	Bit 5	Description
0	0	1 character
0	1	4 characters
1	0	8 characters
1	1	14 characters

Bit 7 Transmit Buffer Load Count

0 = 1 character
1 = 16 characters

See ASYNC_SETDCBINFO for field definitions.

ASYNC_GETDCBINFO (73h) - Error Replacement Character

Error Replacement Character
Returned value. See Note 5 of Function 53h.

ASYNC_GETDCBINFO (73h) - Break Replacement Character

Break Replacement Character
Returned value. See Note 7 of Function 53h.

ASYNC_GETDCBINFO (73h) - XON Character

XON Character
Returned value. See Note 2 of Function 53h.

ASYNC_GETDCBINFO (73h) - XOFF Character

XOFF Character
Returned value. See Note 2 of Function 53h.

ASYNC_GETDCBINFO (73h) - Data Packet Format

Field	Length	C Datatype
Write Timeout	WORD	USHORT
Read Timeout	WORD	USHORT
Flags1	BYTE	BYTE
Flags2	BYTE	BYTE
Flags3	BYTE	BYTE
Error Replacement Character	BYTE	BYTE
Break Replacement Character	BYTE	BYTE
XON Character	BYTE	BYTE
XOFF Character	BYTE	BYTE

Related C Data Structure

The DCBINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

ASYNC_GETDCBINFO (73h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETDCBINFO (73h) - Remarks

If a general failure error is not returned, the physical device driver returns valid information in the Data Packet. The general Device Control Block (DCB) parameter access Functions 53h and 73h are used for:

- Automatic Transmit Flow Control (start/stop transmit when XON/XOFF character received)
- Automatic Receive Flow Control (transmit XON/XOFF when receive buffer fills or empties)
- Determining XON/XOFF characters
- DTR Control mode (enable/disable/input handshaking)
- RTS Control mode (enable/disable/input handshaking/toggling on transmit)
- Output handshaking using CTS/DSR/DCD (control signal determines when to transmit)
- Input sensitivity using DSR (reception of data controlled by DSR)
- Error Replacement character and processing
- Break Replacement character and processing
- Null stripping
- Receive/Transmit Timeout processing
- Extended Hardware Buffering control

To maintain upward compatibility, it is the responsibility of the application to call ASYNC_GETDCBINFO before calling ASYNC_SETDCBINFO. The appropriate information in the returned control block should be modified by the application; then the Function 53h can be performed.

Note: The bit fields that are labeled *reserved* are returned as 0, but applications should not make this assumption or assume that the fourth bit combination will never be returned for DTR Control mode, Read Timeout processing, or Extended Hardware Buffering. Applications should not attempt to manipulate these reserved bits.

ASYNC_GETDCBINFO (73h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETDCBINFO (73h)
Description:
Query Device Control Block (DCB) Parameters

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

ASYNC_GETENHANCEDMODEPARMS (74h) - Query Enhanced M

ASYNC_GETENHANCEDMODEPARMS (74h) - Description

This function returns the Enhanced mode parameters for a COM port. It also returns the state of the current DMA-operation mode and user-requested settings for DMA receive/transmit operations. This IOCtl is not currently supported by COM.SYS.

ASYNC_GETENHANCEDMODEPARMS (74h) - Parameter Packet

None. Packet pointer must be NULL.

ASYNC_GETENHANCEDMODEPARMS (74h) - Enhanced Flags1

Enhanced Flags1

Has the following settings (see Remarks for more information):

Bit 0	Enhanced mode supported by hardware.
Bit 1	Enable the Enhanced mode (default)
Bits 2-3	DMA Receive Operation request. Has the following:

Bit 3	Bit 2	Description
0	0	Disable DMA Receive Capability
0	1	Enable DMA Receive Capability (Default)
1	0	Dedicate a DMA channel to Receive operation
1	1	Reserved

Bits 4-5	DMA Transmit operation request. Has the following:

Bit 5	Bit 4	Description
0	0	Disable DMA Transmit Capability
0	1	Enable DMA Transmit Capability (Default)
1	0	Dedicate a DMA channel to Transmit operation
1	1	Reserved

Bit 6	Receive operation in DMA mode
Bit 7	Transmit operation in DMA mode

ASYNC_GETENHANCEDMODEPARMS (74h) - Reserved

Reserved

Reserved.

ASYNC_GETENHANCEDMODEPARMS (74h) - Data Packet Forma

Field	Length	C Datatype
Enhanced Flags1	BYTE	UCHAR
Reserved	DWORD	ULONG

ASYNC_GETENHANCEDMODEPARMS (74h) - Returns

If the call is made with an invalid Parameter Packet value, a *general failure* error is reported and valid information is not returned in the Data Packet.

ASYNC_GETENHANCEDMODEPARMS (74h) - Remarks

Enhanced Flags1. Bit 1 represents the status of the Enhanced mode. As a default, the Enhanced mode is enabled and this bit is set to 1. Bits 2-5 represent the DMA operation options set by the user that have been executed successfully or the system default, if no user options have been requested so far. The setting of bit 1 is meaningful only when bit 0 is set, and the settings of bits 2-5 are meaningful only when bit 1 is set.

Bit 6 shows the most recent Receive Operation mode. If it is set, the receive operation is in DMA mode. When the system is initialized and no receive operation has been done, the state of this bit is meaningless. Bit 7 shows the most recent Transmit Operation mode. If it is set, the transmit operation is in DMA mode. When the system is initialized and no transmit operation has been done, the state of this bit is meaningless.

ASYNC_GETENHANCEDMODEPARMS (74h) -

Category:
IOCTL_ASYNC (01h)
Function:
ASYNC_GETENHANCEDMODEPARMS (74h)
Description:
Query Enhanced Mode Parameters

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 03h Video Control IOCtl Commands

The following is a summary of the Category 03h Video Control IOCtl Commands:

Function	Description

70h	Allocate an LDT Selector
71h	Deallocate an LDT Selector
72h	Query Pointer Draw Address
73h	Initialize Call Vector Table
74h	ABIOS Pass-Through
75h	Allocate an LDT Selector with Offset
76h	Allocate an LDT Selector with Background Validation Options
7Eh	Allocate Video Buffer
7Fh	Get Address to ROM Font

SCR_ALLOCLDT (70h) - Allocate an LDT Selector

SCR_ALLOCLDT (70h) - Description

Allocates an LDT selector.

SCR_ALLOCLDT (70h) - Parameter Packet Format

Field	Length	C Datatype
32-Bit Physical Address	DWORD	PULONG
Length	WORD	USHORT

Related C Data Structure

The LDTADDRINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

SCR_ALLOCLDT (70h) - Data Packet Format

Field	Length	C Datatype
LDT Selector	WORD	SEL

SCR_ALLOCLDT (70h) - Returns

None.

SCR_ALLOCLDT (70h) - Remarks

This function is supported by the screen device driver. A 32-bit physical address and 16-bit length are passed to the physical device driver. The physical device driver returns an LDT selector for that area of memory.

Read/Write access is granted to any data areas completely contained in the range of addresses from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses from 00000 to FFFFF. Requests for any other data areas result in a return code of ERROR_I24_INVALID_PARAMETER.

If the PhysToUVirt Device helper service returns a nonzero offset along with the requested selector, the selector is deallocated and ERROR_I24_INVALID_PARAMETER is returned. In this case, SCR_ALLOCLDTOFF should be used.

SCR_ALLOCLDT (70h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_ALLOCLDT (70h)
Description:
Allocate an LDT Selector

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCR_DEALLOCLDT (71h) - Deallocate an LDT Selector

SCR_DEALLOCLDT (71h) - Description

This function deallocates an LDT selector.

SCR_DEALLOCLDT (71h) - Parameter Packet Format

Field	Length	C Datatype
LDT Selector	WORD	SEL

SCR_DEALLOCLDT (71h) - Data Packet Format

None.

SCR_DEALLOCLDT (71h) - Returns

None.

SCR_DEALLOCLDT (71h) - Remarks

This function is supported by the screen device driver. An LDT selector, previously created by the screen device driver by way of SCR_ALLOCLDT, is passed in the parameter block. The selector is deallocated.

SCR_DEALLOCLDT (71h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_DEALLOCLDT (71h)
Description:
Deallocate an LDT Selector

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PTR_GETPTRDRAWADDRESS (72h) - Query Pointer Draw Acces

PTR_GETPTRDRAWADDRESS (72h) - Description

This function returns the pointer draw access.

PTR_GETPTRDRAWADDRESS (72h) - Parameter Packet Format

None. Packet pointer must be NULL.

PTR_GETPTRDRAWADDRESS (72h) - Data Packet Format

Field	Length	C Datatype
Return Code	WORD	USHORT
Pointer Draw Routine Entry Point (selector:offset)	DWORD	PFN
Pointer Draw Routine Data Segment Selector	WORD	PCH

Related C Data Structure

The PTRDRAWFUNCTION data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PTR_GETPTRDRAWADDRESS (72h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PTR_GETPTRDRAWADDRESS (72h) - Remarks

This function is used by the mouse subsystem to obtain the entry point address of the pointer draw routine. The pointer draw routine is contained within the physical Pointer Draw device driver. It is called by the physical Mouse device driver to update the pointer image on the screen. The Far-16 address returned by Function 72h is passed by the mouse subsystem to the physical mouse device driver through an IOCtl interface. See Category 07h Mouse Control IOCtl Commands. The physical mouse device driver saves the Far-16 address passed and

uses it when calling the pointer draw routine.

This function is supported by the physical Pointer Draw device driver.

PTR_GETPTRDRAWADDRESS (72h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
PTR_GETPTRDRAWADDRESS (72h)
Description:
Query Pointer Draw Access

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

VID_INITCALLVECTOR (73h) - Initialize Call Vector Table

VID_INITCALLVECTOR (73h) - Description

This function initializes the Call Vector Table.

VID_INITCALLVECTOR (73h) - Parameter Packet Format

None. Packet pointer must be NULL.

VID_INITCALLVECTOR (73h) - Data Packet Format

Field	Length	C Datatype
Far Address of the Call Vector Table	DWORD	PVOID

VID_INITCALLVECTOR (73h) - Returns

None.

VID_INITCALLVECTOR (73h) - Remarks

See the description of the Call Vector Table in the *Display Driver Reference* .

VID_INITCALLVECTOR (73h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
VID_INITCALLVECTOR (73h)
Description:
Initialize Call Vector Table

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCR_ABIOSPASSTHRU (74h) - ABIOS Pass-Through

SCR_ABIOSPASSTHRU (74h) - Description

This function passes an ABIOS request block to Unit 0 of the video device opened by SCREEN$, and returns it to the caller on completion of the request.

SCR_ABIOSPASSTHRU (74h) - Parameter Packet Format

Field	Length
ABIOS Request Block	OTHER

SCR_ABIOSPASSTHRU (74h) - Data Packet Format

None.

SCR_ABIOSPASSTHRU (74h) - Returns

None.

SCR_ABIOSPASSTHRU (74h) - Remarks

This function is supported by the physical screen device driver. The parameter packet is copied to a 64-byte work area that is used as an ABIOS request block. The LID and UNIT fields are overwritten with the Logical ID of the ABIOS video device opened by SCREEN$ and 0, respectively. This request block is used with the DevHlp_ABIOSCall and the resulting request block is copied back to the caller's Parameter Packet.

SCR_ABIOSPASSTHRU (74h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_ABIOSPASSTHRU (74h)
Description:
ABIOS Pass-Through

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCR_ALLOCLDTOFF (75h) - Allocate an LDT Selector with Offset

SCR_ALLOCLDTOFF (75h) - Description

This function allocates an LDT selector with offset.

SCR_ALLOCLDTOFF (75h) - Parameter Packet Format

Field	Length	C Datatype
32-Bit Physical Address	DWORD	PULONG
Length	WORD	USHORT

Related C Data Structure

The LDTADDRINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

SCR_ALLOCLDTOFF (75h) - Data Packet Format

Field	Length	C Datatype
Offset within New LDT Selector	WORD	USHORT
New LDT Selector	WORD	SEL

SCR_ALLOCLDTOFF (75h) - Returns

None.

SCR_ALLOCLDTOFF (75h) - Remarks

This function is supported by the physical screen device driver. A 32-bit physical address and 16-bit length are passed to the device driver. The physical device driver returns an LDT selector and offset for that area of memory.

Read/Write access is granted to any data areas completely contained in the range of addresses from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses from 00000 to FFFFF. Requests for any other data areas result in a return code of ERROR_I24_INVALID_PARAMETER.

SCR_ALLOCLDTOFF (75h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_ALLOCLDTOFF (75h)
Description:
Allocate an LDT Selector with Offset

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

Remarks

SCR_ALLOCLDTBGVAL (76h) - Allocate an LDT Selector with Bacl

SCR_ALLOCLDTBGVAL (76h) - Description

This function allocates an LDT selector with background validation.

SCR_ALLOCLDTBGVAL (76h) - Options

Options

Characteristics of Video RAM (VRAM) address space are customized with the Options parameter. The following values are valid:

0	Make segment readable code; no validation in background
1	Make segment writable data; no validation in background
2	Free selector
3	Make segment readable IOPL code
4	Make segment Read/Write IOPL data
5	Tag selectors Physical Video Buffer (PVB); make segment writable data with validation in background
Other	Reserved

SCR_ALLOCLDTBGVAL (76h) - Parameter Packet Format

Field	Length	C Datatype
32-Bit Physical Address	DWORD	ULONG
Segment Length	WORD	USHORT
Options	WORD	USHORT

SCR_ALLOCLDTBGVAL (76h) - Data Packet Format

Field	Length	C Datatype
LDT Selector:Offset	DWORD	PVOID

SCR_ALLOCLDTBGVAL (76h) - Returns

None.

SCR_ALLOCLDTBGVAL (76h) - Remarks

This function is supported by the physical screen device driver. A 32-bit physical address and 16-bit length are passed to the physical device driver. The physical device driver returns an LDT selector and offset for that area of memory.

Read/Write access is granted to any data areas completely contained in the range of addresses from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses from 00000 to FFFFF. Requests for any other data areas will return an error.

SCR_ALLOCLDTBGVAL (76h) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_ALLOCLDTBGVAL (76h)
Description:
Allocate an LDT Selector with Background Validation

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCR_ALLOCVIDEOBUFFER (7Eh) - Allocate Video Buffer - DBCS

SCR_ALLOCVIDEOBUFFER (7Eh) - Description

This function is used by BVHVGA.DLL and FNTCALLS.DLL.

SCR_ALLOCVIDEOBUFFER (7Eh) - Parameter Packet Format

None.

SCR_ALLOCVIDEOBUFFER (7Eh) - Data Packet Format

Field	Length	C Datatype
Length of Buffer	WORD (In)	USHORT
GDT Selector to Buffer	WORD (Out)	SEL

SCR_ALLOCVIDEOBUFFER (7Eh) - Returns

The error return codes for this function are as follows:

AX= 0 Success

AX ! = 0 Error

SCR_ALLOCVIDEOBUFFER (7Eh) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_ALLOCVIDEOBUFFER (7Eh)
Description:
Allocate Video Buffer - DBCS

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

SCR_GETROMFONTADDR (7Fh) - Get Address to ROM Font - DB

SCR_GETROMFONTADDR (7Fh) - Description

This function is used by FNTCALLS.DLL.

SCR_GETROMFONTADDR (7Fh) - Parameter Packet Format

None.

SCR_GETROMFONTADDR (7Fh) - Data Packet Format

Field	Length	C Datatype
Physical Address	DWORD (In)	ULONG
Length	WORD (In)	USHORT
GDT Selector to ROM Font	WORD (Out)	SEL

SCR_GETROMFONTADDR (7Fh) - Returns

The error return codes for this function are as follows:

AX = 0 Successful.

When the GDT Selector is allocated correctly, the value of the GDT selector allocated will be returned. If the GDT Selector is not allocated correctly, 0 will be returned.

AX != 0 Error

SCR_GETROMFONTADDR (7Fh) -

Category:
IOCTL_SCR_AND_PTRDRAW (03h)
Function:
SCR_GETROMFONTADDR (7Fh)
Description:
Get Address to ROM Font - DBCS

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

Category 04h Keyboard Control IOCtl Commands

The following is a summary of the Category 04h IOCtl Commands:

Function	Description
50h	Set Code Page
51h	Set Input Mode (Default ASCII)
52h	Set Interim Character Flags
53h	Set Shift State
54h	Set Typematic Rate and Delay
55h	Reserved
56h	Set Session Manager Hot Key
57h	Set KCB
58h	Set Code Page Number
59h	Set Read/Peek Notification
5Ah	Alter Keyboard LEDs
5Bh	Reserved
5Ch	Set NLS and Custom Code Page
5Dh	Create a New Logical Keyboard
5Eh	Destroy a Logical Keyboard
71h	Query Input Mode
72h	Query Interim Character Flags
73h	Query Shift State
74h	Read Character Data Records
75h	Peek Character Data Record
76h	Query Session Manager Hot Key
77h	Query Keyboard Type
78h	Query Code Page Number
79h	Translate Scan Code to ASCII
7Ah	Query Keyboard Hardware ID
7Bh	Query Keyboard Code Page Support Information

KBD_SETTRANSTABLE (50h) - Set Code Page

KBD_SETTRANSTABLE (50h) - Description

This function sets the code page.

KBD_SETTRANSTABLE (50h) - Code Page Translation Table

Code Page Translation Table

Has the following format when there are 127 copies of the KeyDef record shown below (includes one for each possible scan code that can be returned from the keyboard). Not all entries are used; unused entries are zero. The entries are in scan code order, based on the remapped scan codes returned by the keyboard controller.

```
XlateTable:
    XHeader          : XHeader
    KeyDef1          : KeyDef
    KeyDef2          : KeyDef
    KeyDef3          : KeyDef
       .                 .
       .                 .
       .                 .
    KeyDef127        : KeyDef
    AccentTbl        : AccentTable
End XlateTable

XHeader:
    XTableID         : WORD [Code PageNumber]
XTableFlags1         : Rec  [Word Width]
                     : The following three bits determine which shift key
                     : or key combination affects CHAR3 of each KeyDef.
    ShiftAlt         : Bit 0 [Use Shift+Alt instead of Ctrl+Alt]
    AltGrafL         : Bit 1 [Use left Alt key as Alt+Graphics]
    AltGrafR         : Bit 2 [Use right Alt key as Alt+Graphics]
    ShiftLock        : Bit 3 [Treat Caps Lock as ShiftLock]
    DefaultTable     : Bit 4 [Default table for the Lang.]
    ShiftToggle      : Bit 5 [Toggle or LatchShiftLock]
                     :       When 1 toggle, else latch
    AccentPass       : Bit 6 [Pass accent and non-accent key through]
                     :       When 1 pass on accent keys and beep, else beep
                     :        only.
                     : The following four bits determine which shift key or
                     : key combination causes Char5 to be used in each
                     :        KeyDef.
    CapsShift        : Bit 7 [Caps+Shift uses CHAR5]
    MachDep          : Bit 8 [Machine-dependent table]
    Reserved         : Bits  9-10
    Reserved         : Bits 11-15
EndRec XtableFlags1



XTableFlags2         : Rec[WORD Width]
    Reserved         : Bits 0-15
EndRec XtableFlags2

    KbdType          : WORD [Keyboard type, see below]
    KbdSubType       : WORD [Reserved]
    XtableLen        : WORD [Length of table]
    EntryCount       : WORD [Number of KeyDef entries]
    EntryWidth       : WORD [Width of KeyDef entries]
    Country          : WORD [Language ID]
    TableTypeID      : WORD [Identifies the table type]
                            1st byte (type)     : 01X  00X
                            2nd byte (sub-type) : 00X  Reserved
    SubCountryID     : 4 Bytes [Sub-language Identifier]
    Reserved         : 8 WORDS [Reserved]
End XHeader

KeyDef = Rec           [127 copies of this record.]
    XlateOp = Rec      [WORD field] [Translate operation specifier.]
        AccentFlags  : 7 Bits [See Notes 1 and 8.]
        KeyType      : 9 bits [See Note 2.]
    Char1            : Byte [Use depends on KeyType, see below.]
    Char2            : Byte [Use depends on KeyType, see below.]
    Char3            : Byte [Use depends on KeyType, see below.]
    Char4            : Byte [Use depends on KeyType, see below.]
    Char5            : Byte [Use depends on KeyType, see below.]
EndRec KeyDef

AccentTable = Rec      [Table of accent key definitions.]
    AccentEntry1     : AccentEntry
    AccentEntry2     : AccentEntry
         .                 .
         .                 .
         .                 .
```

```
    AccentEntry7     : AccentEntry
EndRec AccentTable


AccentEntry = Rec     [Accent entry definition. See Notes 1 and 9.]
    NonAccent       : 2 Bytes [Char/scan code when not used as accent]
    CtlAccent       : 2 Bytes [Char/scan code when used with CTL.]
    AltAccent       : 2 Bytes [Char/scan code when used with Alt.]
    Map1            : 2 Bytes [From char-to-char for translation.]
    Map2            : 2 Bytes   "  "  "  "
      .                   .     "  "  "  "
      .                   .     "  "  "  "
      .                   .     "  "  "  "
    Map20           : 2 Bytes   "  "  "  "
EndRec AccentEntry

TableTypeID
                        1st Byte     2nd Byte
                        type         sub-type
                        00X          Reserved
OS/2                    01X          00X
```

KBD_SETTRANSTABLE (50h) - Parameter Packet Format

Field	Length
Code Page Translation Table	OTHER

KBD_SETTRANSTABLE (50h) - Data Packet Format

None.

KBD_SETTRANSTABLE (50h) - Returns

None.

KBD_SETTRANSTABLE (50h) - Remarks

This request changes the device driver resident code page for the system and updates the zero entry of the Code Page Control Block.

Note 1

The *AccentFlags* field of the KeyDef record has seven flags that are individually set if a corresponding entry in the accent table applies to this scan code. If the key pressed immediately before the current one was an accent key and the bit for that accent is set in the AccentFlags field for the current key, the corresponding AccentTable entry is searched for the replacement character value to use. If no replacement is found and bit 6 of the *XlateFlags1* field is set, the *not-an-accent* beep is sounded and the accent character and current character are passed as two separate characters. Also see Note 8.

Note 2

The *KeyType* field of the KeyDef record currently has the following values defined. The remaining values up to 1Fh are undefined. The effect of each type of shift is defined below. Except where otherwise noted, when no shifts are active, Char1 is the translated character. (See Note 3.) Notice that any of the Alt, Alt+Char, Alt+Shift, or Alt+Gr keys (or all of them) can be present on a keyboard based on the AltGrafL and AltGrafR bits in the XTableFlags1 flag WORD in the table header.

01h AlphaKey. Alphabetical character key:

Shift	Uses Char2. If Caps Lock, uses Char1.
Caps Lock	Uses Char2. If Shift, uses Char1.
Ctrl	Set standard control code for this key's Char1 value. See Note 4.
Alt	Standard extended code. See Note 7.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

02h SpecKey. Special nonalphabetic character key, no Caps Lock or Alt:

Shift	Uses Char2.
Caps Lock	No effect, only depends on Shift, or Ctrl.
Ctrl	See Note 4.
Alt	Marked undefined.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

03h SpecKeyC. Special nonalphabetic character key with Caps Lock. See Note 15.

Shift	Uses Char2. If Caps Lock, uses Char1.
Caps Lock	Uses Char2. If Shift, uses Char1.
Ctrl	See Note 4.
Alt	Uses Char4, if not zero. See Note 7.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

04h SpecKeyA. Special nonalphabetic character key with Alt (no Caps Lock):

Shift	Uses Char2.
Caps Lock	No effect; depends on Shift, Ctrl, or Alt only.
Ctrl	See Notes 5 and 9.
Alt	See Notes 7 and 10.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

05h SpecKeyCA. Special nonalphabetic character key with Caps Lock and Alt:

Shift	Uses Char2. If Caps Lock, uses Char1.
Caps Lock	Uses Char2. If Shift, uses Char1.
Ctrl	See Note 4.
Alt	See Note 7.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

06h FuncKey. Function keys. Char1 = *n* in *Fn*; Char2 ignored. Sets extended codes 58+Char1, if no shift; if F11 or F12, uses 139 and 140.

Shift	Sets extended codes 83+Char1. F11 and F12 use 141 and 142, respectively.
Caps Lock	No effect on function keys.
Ctrl	Sets extended codes 93+Char1. F11 and F12 use 143 and 144, respectively.
Alt	Sets extended codes 103+Char1. F11 and F12 use 145 and 146, respectively.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

07h PadKey. Keypad keys (see Note 5 for definition of Char1). Note that nonshifted use of these keys is fixed to the extended codes:

Shift	Uses Char2, unless Num Lock. See Note 5.
Caps Lock	No effect on pad keys, unless Num Lock. See Note 5.
Ctrl	Sets extended codes. See Note 5.
Alt	Used to build a character. See Note 5.
Alt+Char	Uses Char3, if it is not 0.

Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

08h SpecCtlKey. Special action keys, when used with Ctrl pressed:

Shift	No effect on these keys.
Caps Lock	No effect on these keys.
Ctrl	Uses Char2.
Alt	See Note 7.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

09h PrtSc. Print Screen key; sets Char1 normally (see Note 17):

Shift	Signal the Print Screen function.
Caps Lock	No effect on this key.
Ctrl	Sets extended code and signals the Print Echo function.
Alt	Marked undefined.
Alt+Char	Uses Char3, if it is not 0.
Alt+Shift	Uses Char3, if it is not 0.
Alt+Gr	Uses Char3, if it is not 0.

0Ah SysReq. System Request key; treated like a shift key. See Note 6.

0Bh AccentKey. Keys that affect the next key pressed (also known as *dead keys*). Char1 is an index into the AccentTbl field of the XlateTable, selecting the AccentEntry that corresponds to this key. Char2 and Char3 do the same for the shifted Accent character. See Note 15.

Shift	Uses Char2 to index to applicable AccentEntry.
Caps Lock	No effect on this key.
Ctrl	Uses CtlAccent character from AccentEntry. See Note 8.
Alt	Uses AltAccent character from AccentEntry. See Note 8.
Alt+Char	Uses Char3 to index to applicable AccentEntry.
Alt+Shift	Uses Char3 to index to applicable AccentEntry.
Alt+Gr	Uses Char3 to index to applicable AccentEntry.

Note: Key types 0Ch - 13h set Char1 and Char2 to mask values as defined in Note 6.

0Ch ShiftKeys. Shift or Ctrl key, sets and clears flags. Char1 holds the bits in the lower byte of the shift status WORD to set when the key is down and clear when the key is released. Char2 does the same thing for the upper byte of the shift status WORD unless the secondary key prefix (hex E0) is seen immediately prior to this key, in which case Char3 is used in place of Char2.

0Dh ToggleKey. General toggle key (like Caps Lock). Char1 holds the bits in the lower byte of the shift status WORD to toggle on the first make of the key after it is pressed. Char2 holds the bits in the upper byte of the shift status WORD to set when the key is down and clear when the key is released unless the secondary key prefix (hex E0) is seen immediately prior to this key, in which case Char3 is used in place of Char2.

0Eh AltKey. Treated just like ShiftKeys above, but has its own key type, because when seen, the accumulator used for Alt+PadKey entry is zeroed to prepare such entry (see Note 5). Sometimes this key is treated as the AltC/S/G key (that is, either Alt+Char, Alt+Shift, or Alt+Gr) if one of the AltGraf bits is *on* in XTableFlags1.

0Fh Num Lock. Normally behaves like ToggleKey, but the physical keyboard device driver sets a pause screen indication when this key is used with the Ctrl key pressed. The pause is cleared on the following keystroke if that stroke is a character-generating key.

10h Caps Lock. This key is treated as a type 0Dh toggle key. It has a separate entry here so that it can be processed like a Shift Lock key when that flag is set in the XTableFlags1 WORD in the header. When treated as a Shift Lock, the Caps Lock flag in the shift status WORD is set *on* on any make of this key, and only cleared when the left or right shift key is pressed. Char2 and Char3 are processed the same as ToggleKey.

11h Scroll Lock. Normally behaves like ToggleKey but has a separate entry here. When used with Ctrl, it can be recognized as Ctrl+Break.

12h XShiftKey. Extended Shift Key (for Country support). See Note 9.

13h XToggleKey. Extended Toggle Key (for Country support). See Note 9.

14h SpecKeyCS. Special key 1 for country keyboard processing. See Note 15.

Shift	Uses Char2.
Caps Lock	Uses Char4.
Ctrl	See Note 4.
Alt	See Note 7.

Alt+Char	Uses Char3.
Alt+Shift	Uses Char3.
Alt+Gr	Uses Char3.
Caps+Shift	Uses Char5.

15h SpecKeyAS. Special key 2 for country keyboard processing. See Note 15.

Shift	Uses Char2.
Caps Lock	No effect on this key.
Ctrl	See Note 4.
Alt	Uses Char 4. See Note 14.
Alt+Char	Uses Char 3. See Note 14.
Alt+Shift	Uses Char 3. See Note 14.
Alt+Gr	Uses Char 3. See Note 14.

1Ah Extended Extended key. This corresponds to the BIOS level support provided for INT 16h, Functions 20h, 21h, and 22h.

Shift	Uses Char2.
Caps Lock	No effect on this key.
Ctrl	Uses Char4.
Alt	Uses Char5.
Alt+Char	Uses Char 3, if not *0*.
Alt+Shift	Uses Char 3, if not *0*.
Alt+Gr	Uses Char 3, if not *0*.

16-1FFh Reserved, except for 1Ah, the Extended Extended key (see above).

Note 3

Undefined Character Code. Any key combination that does not fall into any of the defined categories. For example, the Ctrl key pressed along with a key that has no defined control mapping is mapped to the value 0, and the key type is set in the KeyPacket record indicating undefined translation. The KeyPacket record passed to the monitors, if installed, contain the original scan code in the *ScanCode* field and the 0 in the *Character* field for this key. Notice that no character data records with an undefined character code are placed in the keyboard input buffer.

Note 4

Ctrl Key. The six possible situations that can occur when a key is pressed with only the Ctrl+shift key are shown below:

- The key pressed is an AlphaKey character. In this case, the Ctrl plus Char1 combination defines one of the standard defined control codes (all numbers are decimal):

Ctrl-	Mapping	Code Name	Ctrl-	Mapping	Code Name
a	1	SOH	n	14	SO
b	2	STX	o	15	SI
c	3	ETX	p	16	DLE
d	4	EOT	q	17	DC1
e	5	ENQ	r	18	DC2
f	6	ACK	s	19	DC3
g	7	BEL	t	20	DC4
h	8	BS	u	21	NAK
i	9	HT	v	22	SYN
j	10	LF	w	23	ETB
k	11	VT	x	24	CAN
l	12	FF	y	25	EM
m	13	CR	z	26	SUB

 Notice that any key defined as AlphaKey uses the Char1 code value minus 96 (ASCII code for a) plus 1 to set the mapping shown above. Any scan code defined as AlphaKey must assign to Char1 one of the allowed lower case letters.

- The key pressed is a nonalpha character, such as [, but is not an action key, such as Enter, Backspace, or an arrow key. This is a SpecKey[C][A] in the list of key types in the previous example. In this case, with one exception, the mapping is based on the scan code of the key. Though the key can be relabeled, the Ctrl+Char combination is always mapped based on the scan code of the key using the following table (all numbers are decimal):

Scan Code	US Kbd Legend	Mapped Value	Name of New Code
3	2 @	0	Null
7	6 ^	30	RS
12	- _	31	US (see Note below)
26	[{	27	Esc
27	] }	29	GS
43	\ \|	28	FS

Note: The mapping for the hyphen character (-) is the one exception. The scan code for it is ignored; only the ASCII code for hyphen (decimal 45) is looked for in Char1 when mapping the Ctrl+- combination. This is because there can be more than one occurrence of the hyphen (-) key on the keyboard. The Ctrl+- (PadKey minus) combination produces character/scan code values of 00/8Eh, respectively.

- The key pressed is an action key such as Enter, Backspace, or an arrow key. These keys generate special values when used in conjunction with the Ctrl key. Those actions are defined in other notes where they apply. Two particular keys in this category are:

```
Ctrl+Enter     = LF(010)
Ctrl+Backspace = Del(127)
```

- The key pressed is a function key, F1 - F12. See the FuncKey description in Note 2.
- The key pressed is an accent key. See Note 8.
- The key is not defined in conjunction with Ctrl. In this case, the key is treated as undefined, as described in Note 3.

Note 5

PadKey. The pad keys have several uses that depend on various shift states. Some of them are based on their position on the keyboard. Because keyboard layouts change, the hard-coded assumed positions of the keypad keys, with the offset value that must be coded into Char1, are defined below. Any remapping must use the Char1 values shown below for the keys that correspond to the pad keys given by the Legend or Char2 values:

US Kbd Legend	Scan Code	Char1 Required	Char2 US Kbd	With Ctrl
Home 7	71	Decimal 0	ASCII 7	Decimal 119
Up 8	72	" 1	" 8	" 141
PgUp 9	73	" 2	" 9	" 132
-	74	" 3	" -	" 142
Left 4	75	" 4	" 4	" 115
5	76	" 5	" 5	" 143
Right 6	77	" 6	" 6	" 116
+	78	" 7	" +	" 144
End 1	79	" 8	" 1	" 117
Down 2	80	" 9	" 2	" 145
PgDn 3	81	" 10	" 3	" 118
Ins 0	82	" 11	" 0	" 146
Del .	83	" 12	" .	" 147

Notice that when Num Lock is *off*, or if Shift is active and Num Lock *on*, the code returned is the extended code. The code returned corresponds to the Legends above (Home, PgUp, and so forth). When Num Lock is *on*, or if Shift is active and Num Lock is *off*, the code returned is Char2. Notice that the + and - keys also return Char2 when the shift key is down.

When the Alt key is used with the PadKeys, the absolute value of the pressed key (looked up using the required Char1 value) is added to the accumulated value of any of the previous numeric keys pressed, without releasing the Alt key. Before adding the new number to the accumulated value, that accumulation is multiplied by ten, with overflow beyond 255 ignored. When Alt is released, the accumulation becomes a Character code and is passed along with a scan code of zero. Notice that if any key other than the 10 numeric keys is pressed, the accumulated value is reset to zero. When the keypad *, -, or + keys are pressed while the Alt key is down, the extended characters 55, 74, and 78 (decimal) are returned, respectively.

When AltGraphics is used with the PadKeys, the Char3 value is returned if it is nonzero, and if an AltGraf bit is set in XTableFlags1; otherwise, it is treated the same as the Alt key.

On the Enhanced keyboard, the secondary keypad keys return, as an extended character, the scan code of the key plus 80 (decimal) when pressed in conjunction with the Alt key. The secondary / key returns an extended character of 164, when pressed in conjunction with the Alt key.

Note 6

State Key. Each state key entry has Char1, Char2, and Char3 defined as follows:

- **Char1**. A mask to set the appropriate bit in the low byte of the keyboard Shift Flags when the state key is pressed. When the state key is a toggle key, the set bit is toggled each additional time the key is pressed. When the state key is not a toggle key, the set bit is cleared when the key is released.
- **Char2**. A mask to set the appropriate bit in the high byte of the Keyboard Shift Flags when the key is pressed.
- **Char3**. Used in place of Char2 when the secondary key prefix is seen immediately prior to this key.

The masks are shown below (numbers are in hex):

Key	Char1	Char2	Char3
Right Shift	01	00	00

```
Left Shift       02       00       00
Ctrl Shift       04       01       04
Alt Shift        08       02       08
Scroll Lock      10       10       10
Num Lock         20       20       20
Caps Lock        40       40       40
SysReq           00       80       80
```

Notice that the INS key is not treated as a state key, but as a pad key. Also, SysReq is included here because it is treated as a shift key.

Note 7

Alt Character. Most of the keys defined in a category that allows the Alt key (AlphaKey, SpecKeyA, SpecKeyCA) return a value called an *extended character* . This value is a character code of 00H or E0H, with a second byte (using the ScanCode field of the CharData record) defining the extended code. In most cases, this value is the scan code of the key. Since the legend on these keys can be remapped on a foreign language keyboard, the Alt-based extended code is hard to define in a general sense. The following rules are used:

- AlphaKey. The extended code is derived from Char1 (the lower-case character) as it was originally mapped on the PC keyboard. The original scan code value is the extended code that a character returns. These keys can be moved and will still return their original Alt extended codes.
- SpecKeyA and SpecKeyCA. This category is used for all keys that are not an alphabetic character or an action code (like Enter or Backspace, the only exception being the Tab key, which is treated as a character). On foreign keyboards, these keys can be moved around and can have new values assigned to them, such as special punctuation symbols. Therefore, the Alt mappings must be based on the real scan code as any keys defined by the SpecKey_ classification will have only an Alt mapping, if it is in one of the positions defined below. In that case, the Alt extended code is as shown:

```
Scan    US Kbd    Alt          Scan    US Kbd    Alt
Code    Legend    Value        Code    Legend    Value
----    ------    -----        ----    ------    -----
  2      1 !       120          15      Tab       165
  3      2 @       121          26      [ {        26
  4      3 #       122          27      ] }        27
  5      4 $       123          28      Enter      28
  6      5 %       124          39      ; :        39
  7      6 ^       125          40      ' "        40
  8      7 &       126          41      ' ~        41
  9      8 *       127          43      \ |        43   (equals W.T.C. key number 42)
 10      9 (       128          51      , <        51
 11      0 )       129          52      . >        52
 12      - _       130          53      / ?        53
 13      = +       131
```

 The secondary / key returns an extended character of 164 when pressed while Alt is down.

- FuncKey. Defined in Note 2.
- SpecCtlKey. The Alt+ values of the Escape, Backspace, and Enter keys are extended characters equaling 1, 14, and 28 (decimal), respectively.

When AltGraphics is used, the Char3 value is returned if it is nonzero and if an AltGraf bit is set in XTableFlags1. Otherwise, it is treated the same as the Alt key.

Note 8

Accent Key. When an accent key is pressed with Ctrl or Alt, it is treated as a regular key. The character it translates to is the one in the *CtlAccent* or *AltAccent* field of the AccentEntry pointed to by the Char5 value of the KeyDef. If the key being defined has no defined value with Ctrl or Alt, it should have zeros in the field of the undefined combination.

When an accent key is pressed by itself (or with Right Shift, Left Shift, or AltGraphics), it is not translated immediately. The Char1 (or Char2, when Left or Right Shift or AltGraphics is used) index in the KeyDef record is used with the next key received to check if the next key has an accent mapping. If that next key has no mapping for this accent (that is, if it has no bit set in its AccentFlags), or if that next key is not found in this accent's AccentEntry, then the character value in the NonAccent field of the AccentEntry is used as the character to display. It is followed by the translation of that next key after the not-an-accent beep is sounded.

Notice that if a key doesn't change when a Left or Right Shift key is pressed, it should use the same value for Char1 and Char2 so the accent applies in both the shifted and nonshifted cases. If the accent value is undefined when used with a shift key or AltGraphics, the value in Char2 or Char3 should be 0.

Any accent key that doesn't have an Alt or Ctrl mapping should put zeros in the *AltAccent* and *CtlAccent* fields of its AccentEntry. If the value in the table is between 1 and 7, then the key is considered an accent key and further accent key processing is indicated. See Note 1 for more information.

Note 9

Extended State Key. For special Country support, the keyboard device driver maintains another byte of shift status. Key types 12h and 13h

are provided for manipulation of that byte. The other fields of the KeyDef are:

- **Char1**. A mask in which bits that are *on* define the field being used for the Char2 value. Only bits in the NLS shift status byte that correspond to the bits in this byte are altered by the Char2 value.
- **Char2**. For KeyType 12h (Extended Shift), the value to OR into the byte when the make code is seen. Also, the inverted value is ANDed when the break code is seen. For KeyType 13h (Extended Toggle), the value XORed into the byte on each make code seen (break code ignored).
- **Char3**. Use in place of the Char2 when the secondary key prefix (hex E0) is seen immediately prior to this key.

For example, Char1 or Char2 can define single shift status bits to set/clear/toggle. Char2 can be a set of coded bits, delineated by Char1, that are set to a numeric value when the key is pressed and cleared to zero when released (or on the next press, if toggled). The whole byte can be set to Char2 when Char1 has all bits *on*.

Note 10

Space Key. The key treated as the space character should have a flag set in its *AccentFlags* field for each possible accent (that is, for each defined AccentEntry in the AccentTable). And each AccentEntry should have the Space character defined as one of its accented characters, with the translation having the same value as the accent character itself. The reason for this is that, by definition, an Accent Key followed by the space character maps to the accent character alone. If the table is not set up as just described, a not-an-accent beep is sounded whenever the accent key followed by a space is pressed.

Notice that the space key is defined as a SpecKeyA (type 4) because its use, in conjunction with the Alt key, is allowed. In this case, and when used with the Ctrl key, it returns the ASCII space character. This works correctly, except in the case of the diaresis accent (double-dot) in code page 437. The space is treated as an invalid character and the beep result occurs, with the diaresis represented by double quotation marks. The characters displayed depend upon the language in effect when the invalid diaresis is encountered. For some languages, the character substituted is the double-quotation marks; for others, the character used is the F9h character.

Note 11

KbdType identifies the hardware-specific keyboard used by this table. The values and allowable types are the same as those specified in KBD_GETKEYBDTYPE.

Note 12

The DefaultTable flag in XtableFlags1 is used by the KEYB utility in loading code pages when changing from one language to another. It identifies the default code page to KEYB, should KEYB not find one or both CODEPAGE= defined code pages.

Note 13

The Language IDs and Subcountry IDs used are as follows:

Keyboard Layout Country Code	Keyboard Layout SubCountry Code	Country
AR	785	Arabic-speaking
BE	120	Belgium
CF	058	Canadian-French
CS	243	Czech Republic
CS	245	Czech Republic
DK	159	Denmark
SU	153	Finland
FR	120	France
FR	189	France
GR	129	Germany
HE	972	Hebrew-speaking
HU	208	Hungary
IS	197	Iceland
IT	141	Italy
IT	142	Italy
LA	171	Latin-American

		Spanish
NL	143	Netherlands
NO	155	Norway
PL	214	Poland
PO	163	Portugal
SP	172	Spain
SV	153	Sweden
SF	150F	Swiss-French
SG	150G	Swiss-German
TR	179	Turkey
UK	166	United Kingdom
UK	168	United Kingdom
US	103	United States
YU	234	Former Yugoslavia

Note 14

Keytype 15. When the Alt or Alt+Shift keys are pressed, both XlatedChar and XlatedScan in the CharData record will have the same value.

Note 15

If the Char*x* value is in the range of 1-7, then Char*x* identifies an accent key. Otherwise, Char*x* is treated as a valid ASCII character. This does not apply to Ctrl+Char*x* sequences.

Note 16

If Alt+Gr, Alt+Shift, or Alt+Ctrl are pressed, and Char3 is 0, the Alt key is used to translate to a valid result.

Note 17

The * key on the keypad of the Enhanced keyboard, although producing the same scan code/character as that of the IBM Personal Computer AT* keyboard, is treated differently because a dedicated Print Screen key exists on the Enhanced keyboard. The following scan codes/characters are returned by the physical keyboard device driver for the Enhanced keyboard * key on the keypad:

Unshifted	37H/2AH
Shifted	37H/2AH
Ctrl	96H/00
Alt	37H/00

Note 18

Size. The code page described here has the following dimensions:

```
Xlate Header               =    40
127 KeyDefs @ 7 bytes      =   889
7 AccentEntries @ 46 bytes =   322
                               ----
                           1251 bytes
```

If more than 6 AccentEntries are needed, then the following format is used: In the first 6 AccentEntries, the length is set at 20, with unused elements set to zero. For each AccentEntry of 7 and greater, up to 120 element pairs may exist, and the length is dynamic.

For each AccentEntry of 7 and greater, the first byte in the record will contain the LENGTH of the AccentEntry record. The LENGTH value is defined as the total length in bytes of the AccentEntry record including the LENGTH byte.

The record is defined as follows:

```
AccEnt <l,a,b,c,d,e,f,c1,s1,c2,s2..c120,s120>
where....
l is the total length in bytes of the AccEnt including itself.
a &b are the scan code &char to use when the key following this accent
is not affected by the accent so the accent itself must be used.
c &d are the scan code &char to use when Ctl+[accent] is pressed.
```

```
e &f do the same for Alt+[accent].
c1,s1 - c120,s120 are the char/scan code mapping for accented translation.
```

Adding more than 7 accents will make the standard 1251-byte table an extended variable size.

KBD_SETTRANSTABLE (50h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETTRANSTABLE (50h)
Description:
Set Code Page

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETINPUTMODE (51h) - Set Input Mode (Default ASCII)

KBD_SETINPUTMODE (51h) - Description

This function sets the input mode.

KBD_SETINPUTMODE (51h) - Mode

Mode
A 1-byte field containing:

Bit 1xxxxxx1	Shift Report
Bit 0xxxxxx0	ASCII mode
Bit 1xxxxxxx	BINARY mode

KBD_SETINPUTMODE (51h) - Parameter Packet Format

Field	Length	C Datatype
Mode	BYTE	UCHAR

KBD_SETINPUTMODE (51h) - Data Packet Format

None.

KBD_SETINPUTMODE (51h) - Returns

The error return codes for this function are as follows:

13H	INVALID_PARAMETER
0CH	GENERAL_FAILURE

KBD_SETINPUTMODE (51h) - Remarks

This request is used to pass the current input mode to the Keyboard device driver. The Keyboard device driver maintains the mode separately for each session. The caller can interrogate the mode using KBD_GETINPUTMODE. The mode is also returned on every call to KBD_READCHAR, and KBD_PEEKCHAR. The default input mode is ASCII. The physical device driver uses the mode when processing CTL functions and reporting the shift state, if Shift Report is set *on*.

KBD_SETINPUTMODE (51h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETINPUTMODE (51h)
Description:
Set Input Mode (Default ASCII)

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETINTERIMFLAG (52h) - Set Interim Character Flags

KBD_SETINTERIMFLAG (52h) - Description

This function sets the interim character flags.

KBD_SETINTERIMFLAG (52h) - Flag

Flag

A 1-byte field containing bit flags. A bit set to 1 indicates the state listed below:

Bit 7	Interim console flag on.
Bit 6	Reserved. Must equal 0.
Bit 5	Program requested on-the-spot conversion.
Bit 4	Reserved. Must equal 0.
Bit 3	Reserved. Must equal 0.
Bit 2	Reserved. Must equal 0.
Bit 1	Reserved. Must equal 0.
Bit 0	Reserved. Must equal 0.

KBD_SETINTERIMFLAG (52h) - Parameter Packet Format

Field	Length	C Datatype
Flag	BYTE	UCHAR

KBD_SETINTERIMFLAG (52h) - Data Packet Format

None.

KBD_SETINTERIMFLAG (52h) - Returns

The error return code for this function is as follows:

13H INVALID_PARAMETER

KBD_SETINTERIMFLAG (52h) - Remarks

This request is used to set the interim character flags maintained by the physical keyboard device driver. The physical keyboard device driver maintains the flags separately for each session and passes the interim character flags with each character data record to keyboard monitors.

KBD_SETINTERIMFLAG (52h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETINTERIMFLAG (52h)
Description:
Set Interim Character Flags

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETSHIFTSTATE (53h) - Set Shift State

KBD_SETSHIFTSTATE (53h) - Description

This function sets the shift state.

KBD_SETSHIFTSTATE (53h) - Shift State

Shift State
A *WORD* field containing shift states:

High Byte	Has the following settings:	
	Bit 15	SysReq key down
	Bit 14	Caps Lock key down
	Bit 13	NumLock key down
	Bit 12	ScrollLock key down
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down
	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
Low Byte	Has the following:	
	Bit 7	Insert *on*
	Bit 6	Caps Lock *on*
	Bit 5	NumLock *on*
	Bit 4	ScrollLock *on*
	Bit 3	Either Alt key down
	Bit 2	Either Ctrl key down
	Bit 1	Left Shift key down
	Bit 0	Right Shift key down

KBD_SETSHIFTSTATE (53h) - NLS

NLS
A byte field containing NLS shift status.

KBD_SETSHIFTSTATE (53h) - Parameter Packet Format

Field	Length	C Datatype
Shift State	WORD	USHORT
NLS	BYTE	BYTE

Related C Data Structure

The SHIFTSTATE data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

KBD_SETSHIFTSTATE (53h) - Data Packet Format

None.

KBD_SETSHIFTSTATE (53h) - Returns

None.

KBD_SETSHIFTSTATE (53h) - Remarks

This request is used to set the current shift state for the keyboard. The physical keyboard device driver maintains the shift state separately for each logical keyboard. Notice that this call overrides the Shift State set by previous shift keystrokes. Also, the Shift State set by this function code is overridden by any subsequent shift keystrokes. This shift state is inserted into the Character Data Record that is built for each incoming keystroke.

KBD_SETSHIFTSTATE (53h) -

Category:
IOCTL_KEYBOARD (04h)
Function:

KBD_SETSHIFTSTATE (53h)

Description:

Set Shift State

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETTYPAMATICRATE (54h) - Set Typematic Rate and Delay

KBD_SETTYPAMATICRATE (54h) - Description

This function sets typematic rate and delay.

KBD_SETTYPAMATICRATE (54h) - Delay

Delay

Specifies the typematic delay in milliseconds. A value greater than the maximum value defaults to the maximum value.

KBD_SETTYPAMATICRATE (54h) - Rate

Rate

Specifies the typematic rate in characters per second. A value greater than the maximum value defaults to the maximum value.

KBD_SETTYPAMATICRATE (54h) - Parameter Packet Format

Field	Length	C Datatype
Delay	WORD	USHORT
Rate	WORD	USHORT

Related C Data Structure

The RATEDELAY data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

KBD_SETTYPAMATICRATE (54h) - Data Packet Format

None.

KBD_SETTYPAMATICRATE (54h) - Returns

None.

KBD_SETTYPAMATICRATE (54h) - Remarks

This request is used to set the keyboard typematic rate and delay to the values specified in the request.

KBD_SETTYPAMATICRATE (54h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETTYPAMATICRATE (54h)
Description:
Set Typematic Rate and Delay

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETSESMGRHOTKEY (56h) - Set Session Manager Hot Key

KBD_SETSESMGRHOTKEY (56h) - Description

This function sets the Session Manager hot key.

KBD_SETSESMGRHOTKEY (56h) - State

State

Has the following settings:

High Byte	Bit settings are as follows:	
	Bit 15	Reserved = *0*
	Bit 14	Reserved = *0*
	Bit 13	Reserved = *0*
	Bit 12	Reserved = *0*
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down
	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
Low Byte	Bit settings are as follows:	
	Bit 7	Reserved = 0
	Bit 6	Reserved = 0
	Bit 5	Reserved = 0
	Bit 4	Reserved = 0
	Bit 3	Reserved = 0
	Bit 2	Reserved = 0
	Bit 1	Left Shift key down
	Bit 0	Right Shift key down

KBD_SETSESMGRHOTKEY (56h) - Make Code

Make Code

The scan code of the hot key, Make

KBD_SETSESMGRHOTKEY (56h) - Break Code

Break Code

The scan code of the hot key, Break

KBD_SETSESMGRHOTKEY (56h) - Hot Key ID

Hot Key ID

The Hot Key ID. Value is set by the caller. Notice that ID value FFFFh is reserved and must not be used as a Hot Key ID. See **Remarks**.

A maximum of two of the above bit selections can be selected for a given hot key definition. If more than two bits are selected or if a reserved bit is selected, the result is an INVALID_PARAMETER error code returned to the caller.

KBD_SETSESMGRHOTKEY (56h) - Parameter Packet Format

Field	Length	C Datatype
State	WORD	USHORT
Make Code	BYTE	UCHAR
Break Code	BYTE	UCHAR
Hot Key ID	WORD	USHORT

Related C Data Structure

The HOTKEY data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

KBD_SETSESMGRHOTKEY (56h) - Data Packet Format

None.

KBD_SETSESMGRHOTKEY (56h) - Returns

The error return codes for this function are as follows:

03H	BAD_COMMAND
13H	INVALID_PARAMETER

KBD_SETSESMGRHOTKEY (56h) - Remarks

This request is used by the Session Manager to set a list of keyboard hot keys, searched for by the physical keyboard device driver. Up to 16 hot keys can be defined by the Session Manager for handling by the physical keyboard device driver. The new hot key is global (that is, it applies to all sessions). A hot key can be redefined by calling this function with the same Hot Key ID.

The combination of shift flags in the first WORD and scan codes in the second allows the Session Manager to set hot key combinations such as Alt+Esc. The hot key is triggered on detection of the scan code for the hot key break.

Note: If a DOS application has claimed hardware INT 9 or INT 50, the hot key is triggered on detection of the break scan code for the required shift key.

Hot key definition requests that have identical data definitions (shift state, make, and break scan codes) but different Hot Key IDs result in an INVALID_PARAMETER error code returned to the caller. Attempts to define a new or unique hot key when the maximum number is currently active also result in an INVALID_PARAMETER error code returned to the caller.

This IOCtl is successful only if performed by the process that initially invoked it, unless it is called with a Hot Key ID of *-1*. A Hot Key ID with a value of *-1* acts like a toggle. The first time a call is made with *-1*, the Ctrl+Alt+Del keystrokes are disabled. The second time a call is made with *-1*, the keystroke sequences are enabled.

The caller of this interface must use caution in the key combinations selected because hot key definitions become system property, and any resulting characters normally produced by that key combination will no longer be available to applications.

KBD_SETSESMGRHOTKEY (56h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETSESMGRHOTKEY (56h)
Description:
Set Session Manager Hot Key

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETKCB (57h) - Set Keyboard Control Block (KCB)

KBD_SETKCB (57h) - Description

This function sets the Keyboard Control Block (KCB).

KBD_SETKCB (57h) - KCB Handle

KCB Handle
The handle identifying the logical keyboard's KCB. A value of 0 in this parameter indicates the default keyboard.

KBD_SETKCB (57h) - Parameter Packet Format

Field	Length	C Datatype
KCB Handle	WORD	HKBD

KBD_SETKCB (57h) - Data Packet Format

None.

KBD_SETKCB (57h) - Returns

The error return codes for this function are as follows:

03H	BAD_COMMAND
0CH	GENERAL_FAILURE

KBD_SETKCB (57h) - Remarks

This request binds the specified logical keyboard (KCB) to the physical keyboard for this session. This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

KBD_SETKCB (57h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETKCB (57h)
Description:
Set Keyboard Control Block (KCB)

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETCP (58h) - Set Code Page Number

KBD_SETCP (58h) - Description

This function sets the code page number.

KBD_SETCP (58h) - Code Page Number

Code Page Number
1-WORD containing the current code page number.

KBD_SETCP (58h) - Code Page Layer

Code Page Layer
Has the following settings:

0	Primary Code Page Layer
1	Secondary Code Page Layer

KBD_SETCP (58h) - Parameter Packet Format

Field	Length	C Datatype
Code Page Number	WORD	USHORT
Code Page Layer	WORD	USHORT

KBD_SETCP (58h) - Data Packet Format

None.

KBD_SETCP (58h) - Returns

The error return codes for this function are as follows:

0CH	GENERAL_FAILURE
13H	INVALID_PARAMETER

KBD_SETCP (58h) - Remarks

Sets the code page translation table, used by the current KCB, to the code page translation table identified by the input parameter. This IOCtl can be called from a DOS session.

KBD_SETCP (58h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETCP (58h)
Description:
Set Code Page Number

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETREADNOTIFICATION (59h) - Set Read/Peek Notification

KBD_SETREADNOTIFICATION (59h) - Description

This function sets read/peek notification.

KBD_SETREADNOTIFICATION (59h) - Session Number

Session Number
Indicates the session to be enabled or disabled.

KBD_SETREADNOTIFICATION (59h) - Read/Peek Notification Flag

Read/Peek Notification Flags
Contains the following flags:

Bits 15-1	Reserved. Set to 0.	
Bit 0	Mask indicating whether to enable or disable Read/Peek notification:	
	0	Disable Read/Peek notification
	1	Enable Read/Peek notification

KBD_SETREADNOTIFICATION (59h) - Parameter Packet Format

Field	Length	C Datatype
Session Number	WORD	USHORT
Read/Peek Notification Flags	WORD	USHORT

KBD_SETREADNOTIFICATION (59h) - Data Packet Format

None.

KBD_SETREADNOTIFICATION (59h) - Returns

The error return codes for this function are as follows:

0CH	BAD_COMMAND
13H	INVALID_PARAMETER

KBD_SETREADNOTIFICATION (59h) - Remarks

Sets the DDFlags of a key packet, instructing the keyboard device driver to send a special monitor packet down the monitor chain. This packet is sent on every Read request seen by the physical keyboard device driver for a particular screen group, and continues until the screen group is terminated or a subsequent call disables Read Notification.

The Process ID (PID) of the first caller of this IOCtl in the system is saved by the physical Keyboard device driver. Subsequent invocations of this IOCtl by a different PID result in an UNKNOWN_COMMAND error.

KBD_SETREADNOTIFICATION (59h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETREADNOTIFICATION (59h)
Description:
Set Read/Peek Notification

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_ALTERKBDLED (5Ah) - Alter Keyboard LEDs

KBD_ALTERKBDLED (5Ah) - Description

This function alters the keyboard LEDs.

KBD_ALTERKBDLED (5Ah) - State of Each LED

State of Each LED

A bit mask indicating the state of the new LED indicators. A set bit means a particular LED is turned *on*. A clear bit indicates the LED is turned *off*. The bit mask is defined as follows:

Bits 15-3	Reserved. Set to 0.
Bit 2	If set, Caps Lock LED turned *on*. If clear, Caps Lock LED turned *off*.
Bit 1	If set, Num Lock LED turned *on*. If clear, Num Lock LED turned *off*.
Bit 0	If set, Scroll Lock LED turned *on*. If clear, Scroll Lock LED turned *off*.

If any of the reserved bits are set, an INVALID_PARAMETER error is returned.

KBD_ALTERKBDLED (5Ah) - Parameter Packet Format

Field	Length	C Datatype
State of Each LED	WORD	USHORT

KBD_ALTERKBDLED (5Ah) - Data Packet Format

None.

KBD_ALTERKBDLED (5Ah) - Returns

The error return codes for this function are as follows:

03H	BAD_COMMAND
13H	INVALID_PARAMETER

KBD_ALTERKBDLED (5Ah) - Remarks

Alters the keyboard LEDs without changing the operational state of the keyboard's mode of scan code generation. This IOCtl is reserved for use by the Presentation Manager session system and cannot be called by applications. Requests from applications return an UNKNOWN_COMMAND error.

Because it is not possible to separate the keyboard's LED and operational state for the 88/89-Key Enhanced Keyboard, the physical Keyboard device driver performs no operations for IOCtl requests received when the 88/89-Key Enhanced keyboard is the attached system keyboard.

KBD_ALTERKBDLED (5Ah) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_ALTERKBDLED (5Ah)
Description:
Alter Keyboard LEDs

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_SETNLS (5Ch) - Set NLS and Custom Code Page

KBD_SETNLS (5Ch) - Description

This function sets the National Language Support (NLS) and custom code page.

KBD_SETNLS (5Ch) - Code Page Translation Table Pointer

Code Page Translation Table Pointer
The selector:offset pointing to the Code Page Translation Table. Notice that the physical keyboard device driver does not perform machine model or submodel determination to validate this translate table address for a given machine or keyboard. This determination is the responsibility of the keyboard subsystem and system initialization routines.

KBD_SETNLS (5Ch) - Code Page Number

Code Page Number
1-WORD identifying the code page number.

KBD_SETNLS (5Ch) - Index to Load

Index to Load

1-WORD identifying the number of the Code Page Control Block to load (1 or 2). A -1 indicates a custom code page for which the segment containing the custom code page is locked. This option is not valid for a DOS session.

KBD_SETNLS (5Ch) - Parameter Packet Format

Field	Length	C Datatype
Code Page Translation Table Pointer	DWORD	ULONG
Code Page Number	WORD	USHORT
Index to Load	WORD	SHORT

KBD_SETNLS (5Ch) - Data Packet Format

None.

KBD_SETNLS (5Ch) - Returns

The error return codes for this function are as follows:

0CH	GENERAL_FAILURE
13H	INVALID_PARAMETER

KBD_SETNLS (5Ch) - Remarks

This request installs one of two possible Code Page Translation Tables into the device driver. This IOCtl updates the number 1 (or 2) entry of the Code Page Control Block. Entry zero is the device-driver-resident Code Page Translation Table.

Notice that this IOCtl is similar to KBD_SETTRANSTABLE, except that different entries in the Code Page Control Block are updated. This IOCtl can be called from a DOS session.

KBD_SETNLS (5Ch) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_SETNLS (5Ch)
Description:
Set NLS and Custom Code Page

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_CREATE (5Dh) - Create New Logical Keyboard

KBD_CREATE (5Dh) - Description

This function creates a new logical keyboard.

KBD_CREATE (5Dh) - KCB ID

KCB ID
A WORD containing a handle (from DosOpenHandle) used to identify the new logical keyboard.

KBD_CREATE (5Dh) - Code Page ID

Code Page ID
A WORD containing the desired initial code page to be used for the new logical keyboard.

KBD_CREATE (5Dh) - Parameter Packet Format

Field	Length	C Datatype
KCB ID	WORD	HKBD
Code Page ID	WORD	USHORT

KBD_CREATE (5Dh) - Data Packet Format

None.

KBD_CREATE (5Dh) - Returns

The error return codes for this function are as follows:

03H	BAD_COMMAND
0CH	GENERAL_FAILURE
13H	INVALID_PARAMETER

KBD_CREATE (5Dh) - Remarks

The desired initial code page should be either the default code page for the system or one of the two optional code pages that can be prepared in the CODEPAGE= statement in the CONFIG.SYS file.

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

KBD_CREATE (5Dh) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_CREATE (5Dh)
Description:
Create New Logical Keyboard

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_DESTROY (5Eh) - Destroy Logical Keyboard

KBD_DESTROY (5Eh) - Description

This function destroys an existing logical keyboard.

KBD_DESTROY (5Eh) - KCB ID

KCB ID
A WORD containing a unique value used to identify the logical keyboard. A zero indicates the default keyboard.

KBD_DESTROY (5Eh) - Parameter Packet Format

Field	Length	C Datatype
KCB ID	WORD	HKBD

KBD_DESTROY (5Eh) - Data Packet Format

None.

KBD_DESTROY (5Eh) - Returns

The error return code for this function is as follows:

0CH GENERAL_FAILURE

KBD_DESTROY (5Eh) - Remarks

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

KBD_DESTROY (5Eh) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_DESTROY (5Eh)
Description:
Destroy Logical Keyboard

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_GETINPUTMODE (71h) - Query Input Mode

KBD_GETINPUTMODE (71h) - Description

This function returns the input mode.

KBD_GETINPUTMODE (71h) - Mode

Mode

A 1-byte field containing one of the following values:

Bit 1xxxxxx1	Shift report
Bit 0xxxxxx0	ASCII mode
Bit 1xxxxxxx	BINARY mode

KBD_GETINPUTMODE (71h) - Parameter Packet Format

Field	Length	C Datatype
Mode	BYTE	UCHAR

KBD_GETINPUTMODE (71h) - Data Packet Format

None.

KBD_GETINPUTMODE (71h) - Returns

None.

KBD_GETINPUTMODE (71h) - Remarks

This request is used to obtain the input mode of the session of the currently active process. The input mode can be set with KBD_SETINPUTMODE. The input mode is meaningful for Ctrl+C, Ctrl+P, Ctrl+S, Ctrl+Break, Ctrl+ScrollLock, and Ctrl+PrtSc processing only.

KBD_GETINPUTMODE (71h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_GETINPUTMODE (71h)
Description:
Query Input Mode

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_GETINTERIMFLAG (72h) - Query Interim Character Flags

KBD_GETINTERIMFLAG (72h) - Description

This function returns the interim character flags.

KBD_GETINTERIMFLAG (72h) - Parameter Packet Format

None.

KBD_GETINTERIMFLAG (72h) - Flags

Flags
A 1-byte field containing flag bits. A bit set to 1 indicates the states listed below:

Bit 7	Interim console flag *on*
Bit 6	Reserved = 0
Bit 5	Program requested on-the-spot conversion
Bit 4	Reserved = 0
Bit 3	Reserved = 0
Bit 2	Reserved = 0
Bit 1	Reserved = 0
Bit 0	Reserved = 0

KBD_GETINTERIMFLAG (72h) - Data Packet Format

Field	Length	C Datatype
Flags	BYTE	UCHAR

KBD_GETINTERIMFLAG (72h) - Returns

None.

KBD_GETINTERIMFLAG (72h) - Remarks

This request is used to obtain the interim character flags maintained by the physical keyboard device driver.

KBD_GETINTERIMFLAG (72h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_GETINTERIMFLAG (72h)
Description:
Query Interim Character Flags

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_GETSHIFTSTATE (73h) - Query Shift State

KBD_GETSHIFTSTATE (73h) - Description

This function returns the shift state.

KBD_GETSHIFTSTATE (73h) - Parameter Packet Format

None.

KBD_GETSHIFTSTATE (73h) - Shift State

Shift State

A WORD field containing shift states:

High Byte	Has the following settings:	
	Bit 15	SysReq key down
	Bit 14	Caps Lock key down
	Bit 13	NumLock key down
	Bit 12	ScrollLock key down
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down
	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
Low Byte	Has the following:	
	Bit 7	Insert *on*
	Bit 6	Caps Lock *on*
	Bit 5	NumLock *on*
	Bit 4	ScrollLock *on*
	Bit 3	Either Alt key down
	Bit 2	Either Ctrl key down
	Bit 1	Left Shift key down
	Bit 0	Right Shift key down

KBD_GETSHIFTSTATE (73h) - NLS

NLS

A byte field containing NLS shift status.

KBD_GETSHIFTSTATE (73h) - Data Packet Format

Field	Length	C Datatype
Shift State	WORD	USHORT
NLS	BYTE	BYTE

Related C Data Structure

The SHIFTSTATE data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

KBD_GETSHIFTSTATE (73h) - Returns

None.

KBD_GETSHIFTSTATE (73h) - Remarks

This request obtains the shift state of the session of the currently active process. The shift state is set by incoming key strokes and by calling KBD_SETSHIFTSTATE.

KBD_GETSHIFTSTATE (73h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_GETSHIFTSTATE (73h)
Description:
Query Shift State

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_READCHAR (74h) - Read Character Data Records

KBD_READCHAR (74h) - Description

This function reads Character Data Records.

KBD_READCHAR (74h) - Transfer Count

Transfer Count

A 1-WORD field containing the record transfer count. The sign bit of this WORD is set to request one of the following actions:

0	Wait for the requested number of keystrokes to become available. The physical device driver blocks the requestor until all requested Character Data Records are available and are transferred to the caller.
1	Do not wait for the requested number of keystrokes to become available. In this case, all characters currently available are transferred up to the requested transfer count.

KBD_READCHAR (74h) - Parameter Packet Format

Field	Length	C Datatype
Transfer Count	WORD	SHORT

KBD_READCHAR (74h) - CharData Record

CharData Record

The character data structure of the API function KbdCharIn is shown below:

ASCIICharCode (UCHAR)	ASCII character code. The scan code received from the keyboard is translated to the ASCII character code.	
ScanCode (UCHAR)	Code received from the keyboard. Scan code received from the keyboard is translated to the ASCII character code.	
Status (UCHAR)	State of the keystroke event:	
	Bits 7-6	Has the following values:
		00 Undefined
		01 Final character; interim character flag is turned *off*
		10 Interim ract

			er
		11	Final character; interim character flag is turned *on*.
	Bit 5	If set to 1, immediate conversion requested	
	Bits 4-2	Reserved	
	Bit 1	Has the following values:	
		0	Scan code is a character
		1	Scan code is not a character; instead it is an extended key code from the keyboard.
	Bit 0	If set to 1, shift status returned without a character.	
Reserved (UCHAR)	NLS shift status. Reserved; must be set to 0.		
ShiftKeyStat (USHORT)	Shift key status. Values are:		
	Bit 15	SysReq key down	
	Bit 14	Caps Lock key down	
	Bit 13	NumLock key down	
	Bit 12	Scroll Lock key down	
	Bit 11	Right Alt key down	
	Bit 10	Right Ctrl key down	
	Bit 9	Left Alt key down	
	Bit 8	Left Ctrl key down	
	Bit 7	Insert *on*	
	Bit 6	Caps Lock *on*	
	Bit 5	NumLock *on*	
	Bit 4	Scroll Lock *on*	
	Bit 3	Either Alt key down	
	Bit 2	Either Ctrl key down	
	Bit 1	Left Shift key down	
	Bit 0	Right Shift key down	
Time (ULONG)	Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.		

KBD_READCHAR (74h) - Data Packet Format

Field	Length	C Datatype
CharData Record	10 BYTES	KBDKEYINFO

KBD_READCHAR (74h) - Returns

The error return codes for this function are as follows:

03H	BAD_COMMAND
11H	CHAR_CALL_INTERRUPTED
13H	INVALID_PARAMETER

KBD_READCHAR (74h) - Remarks

This request is used to obtain one or more character data records from the Keyboard Input buffer (KIB) for the session of the currently active process. Notice that if shift report is *on*, then the CharData Record might not contain a character; instead, it could contain a shift state change in the Shift Status field.

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

KBD_READCHAR (74h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_READCHAR (74h)
Description:
Read Character Data Records

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_PEEKCHAR (75h) - Peek Character Data Record

KBD_PEEKCHAR (75h) - Description

This function peeks at a Character Data Record.

KBD_PEEKCHAR (75h) - Status

Status

A 1-WORD field that contains 0 to indicate no keystroke is available or 1 to indicate that a Character data Record is being returned. The sign bit is set to 0 if the current input mode is ASCII or 1 if the input mode is binary.

KBD_PEEKCHAR (75h) - Parameter Packet Format

Field	Length	C Datatype
Status	WORD	SHORT

KBD_PEEKCHAR (75h) - CharData Record

CharData Record

The character data structure of the API function KbdCharIn is shown below:

ASCIICharCode (UCHAR)	ASCII character code. The scan code received from the keyboard is translated to the ASCII character code.
ScanCode (UCHAR)	Code received from the keyboard. Scan code received from the keyboard is translated to the ASCII character code.
Status (UCHAR)	State of the keystroke event:

Bits 7-6	Has the following values:
00	Undefined
01	Final character; interim character flag

		is turned *off*
	10	Interim character
	11	Final character; interim character flag is turned *on*.
Bit 5	If set to 1, immediate conversion requested	
Bits 4-2	Reserved	
Bit 1	Has the following values:	
	0	Scan code is a character
	1	Scan code is not a character; instead it is an extended key code from the keyboard.
Bit 0	If set to 1, shift status returned without a character.	

Reserved (UCHAR)

NLS shift status. Reserved; must be set to 0.

ShiftKeyStat (USHORT)

Shift key status. Values are:

Bit 15	SysReq key down
Bit 14	Caps Lock key down
Bit 13	NumLock key down
Bit 12	Scroll Lock key down
Bit 11	Right Alt key down
Bit 10	Right Ctrl key down
Bit 9	Left Alt key down
Bit 8	Left Ctrl key down
Bit 7	Insert *on*
Bit 6	Caps Lock *on*
Bit 5	NumLock *on*

Bit 4	Scroll Lock *on*
Bit 3	Either Alt key down
Bit 2	Either Ctrl key down
Bit 1	Left Shift key down
Bit 0	Right Shift key down

Time (ULONG)	Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.

KBD_PEEKCHAR (75h) - Data Packet Format

Field	Length	C Datatype
CharData Record	10 BYTES	KBDKEYINFO

KBD_PEEKCHAR (75h) - Returns

The error return codes for this function are as follows:

03H	BAD_COMMAND
11H	CHAR_CALL_INTERRUPTED
13H	INVALID_PARAMETER

KBD_PEEKCHAR (75h) - Remarks

This request is used to obtain one or more character data records from the Keyboard Input buffer (KIB) for the session of the currently active process. The CharData Record is not removed from the KIB. Notice that if shift report is *on*, then the CharData Record might not contain a character; instead, it could contain a shift state change in the Shift Status field.

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

KBD_PEEKCHAR (75h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_PEEKCHAR (75h)
Description:
Peek Character Data Record

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_GETSESMGRHOTKEY (76h) - Query Session Manager Hot K

KBD_GETSESMGRHOTKEY (76h) - Description

This function returns the Session Manager hot key.

KBD_GETSESMGRHOTKEY (76h) - Type

Type

A 1-WORD subtype indicating the type of information to return:

0	Return the maximum number of hot keys the physical keyboard device driver can support. This number is returned in the Parameter Packet.
1	Return the number of hot keys currently defined for the system (this number is returned in the Parameter Packet), and return the Session Manager hot key information for each number (this information is returned in the Data Packet buffer).

If the parameter list on entry is 1, one or more hot key data structures are returned in the Data Packet format.

KBD_GETSESMGRHOTKEY (76h) - Parameter Packet Format

Field	Length	C Datatype
Type	WORD	USHORT

KBD_GETSESMGRHOTKEY (76h) - State

State

Has the following settings:

High Byte	Bit settings are as follows:	
	Bit 15	Reserved = 0
	Bit 14	Reserved = 0
	Bit 13	Reserved = 0
	Bit 12	Reserved = 0
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down

	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
Low Byte	Bit settings are as follows:	
	Bit 7	Reserved = 0
	Bit 6	Reserved = 0
	Bit 5	Reserved = 0
	Bit 4	Reserved = 0
	Bit 3	Reserved = 0
	Bit 2	Reserved = 0
	Bit 1	Left Shift key down
	Bit 0	Right Shift key down

KBD_GETSESMGRHOTKEY (76h) - Make Code

Make Code
The scan code of the hot key, Make

KBD_GETSESMGRHOTKEY (76h) - Break Code

Break Code
The scan code of the hot key, Break

KBD_GETSESMGRHOTKEY (76h) - Hot Key ID

Hot Key ID
The Hot Key ID. Value is set by the caller. Notice that ID value FFFFh is reserved and should not be used.

A maximum of two of the above bit selections can be chosen for a given hot key definition. If more than two bits are selected or if a reserved bit is selected, the result is an INVALID_PARAMETER error code returned to the caller.

KBD_GETSESMGRHOTKEY (76h) - Data Packet Format

Field	Length	C Datatype
State	WORD	USHORT
Make Code	BYTE	UCHAR
Break Code	BYTE	UCHAR
Hot Key ID	WORD	USHORT

Related C Data Structure

The HOTKEY data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

KBD_GETSESMGRHOTKEY (76h) - Returns

The error return code for this function is as follows:

13H INVALID_PARAMETER

KBD_GETSESMGRHOTKEY (76h) - Remarks

This request obtains the scan code the physical keyboard device driver uses as the Session Manager hot key. It should first be called with parameter list subtype=*0* to determine the maximum number of hot keys the physical device driver can support. The value returned is used to determine the required size of the data buffer on a subsequent call to return the hot key data structures (parameter list subtype=*1*).

KBD_GETSESMGRHOTKEY (76h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_GETSESMGRHOTKEY (76h)
Description:
Query Session Manager Hot Key

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_GETKEYBDTYPE (77h) - Query Keyboard Type

KBD_GETKEYBDTYPE (77h) - Description

This function returns a keyboard type.

KBD_GETKEYBDTYPE (77h) - Parameter Packet Format

None.

KBD_GETKEYBDTYPE (77h) - Keyboard Type

Keyboard Type

A 1-WORD field containing:

High Byte	Reserved. Set to 0.	
Low Byte	Has the following bit values:	
	00h	Personal Computer AT keyboard
	01h	Enhanced Keyboard
	02H-FFh	Reserved. Set to 0

KBD_GETKEYBDTYPE (77h) - Data Packet Format

Field	Length	C Datatype
Keyboard Type	WORD	USHORT
Reserved.	DWORD	ULONG

KBD_GETKEYBDTYPE (77h) - Returns

None.

KBD_GETKEYBDTYPE (77h) - Remarks

This request returns the keyboard type. The 101- and 102-Key Enhanced keyboards, 88- and 89-Key Enhanced keyboards, 122-Key Enhanced keyboards, and MFI keyboards all return the value 01H to indicate an Enhanced keyboard is attached. For specific keyboard Hardware IDs, see KBD_QUERYKBDHARDWAREID.

KBD_GETKEYBDTYPE (77h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_GETKEYBDTYPE (77h)
Description:
Query Keyboard Type

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_GETCODEPAGEID (78h) - Query Code Page Number

KBD_GETCODEPAGEID (78h) - Description

This function returns the code page number.

KBD_GETCODEPAGEID (78h) - Code Page Number

Code Page Number

A WORD containing the current code page number in use. Values other than Code Page IDs that can be returned are as follows:

0	Indicates that PC US 437 is used.
-1	Indicates that a custom code page is installed.

KBD_GETCODEPAGEID (78h) - Parameter Packet Format

Field	Length	C Datatype
Code Page Number	WORD	SHORT
Reserved. Set to 0.	WORD	USHORT

KBD_GETCODEPAGEID (78h) - Data Packet Format

None.

KBD_GETCODEPAGEID (78h) - Returns

None.

KBD_GETCODEPAGEID (78h) - Remarks

This request returns the code page in use by the current KCB. This IOCtl can be called from a DOS session.

KBD_GETCODEPAGEID (78h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_GETCODEPAGEID (78h)
Description:
Query Code Page Number

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_XLATESCAN (79h) - Translate Scan Code to ASCII

KBD_XLATESCAN (79h) - Description

This function translates scan code to ASCII.

KBD_XLATESCAN (79h) - Code Page Number

Code Page Number
1-WORD containing the current code page number.

KBD_XLATESCAN (79h) - Parameter Packet Format

Field	Length	C Datatype
Code Page Number	WORD	USHORT
Reserved. Set to 0.	WORD	USHORT

Related C Data Structure

The CPID data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

KBD_XLATESCAN (79h) - CharData Record

CharData Record

The character data structure of the API function KbdCharIn is shown below:

ASCIICharCode (UCHAR) — ASCII character code. The scan code received from the keyboard is translated to the ASCII character code.

ScanCode (UCHAR) — Code received from the keyboard. Scan code received from the keyboard is translated to the ASCII character code.

Status (UCHAR) — State of the keystroke event:

Bits 7-6 — Has the following values:

00 — Undefined.

01 — Final character; interim character flag is turned *off*.

10 — Interim character.

11 — Final character; interim character flag is turned

on .

	Bit 5	If set to 1, immediate conversion requested.
	Bits 4-2	Reserved.
	Bit 1	Has the following values: 0 Scan code is a character 1 Scan code is not a character; instead it is an extended key code from the keyboard.
	Bit 0	If set to 1, shift status returned without a character.
Reserved (UCHAR)	NLS shift status. Reserved; must be set to 0.	
ShiftKeyStat (USHORT)	Shift key status. Values are:	
	Bit 15	SysReq key down
	Bit 14	Caps Lock key down
	Bit 13	NumLock key down
	Bit 12	Scroll Lock key down
	Bit 11	Right Alt key down
	Bit 10	Right Ctrl key down
	Bit 9	Left Alt key down
	Bit 8	Left Ctrl key down
	Bit 7	Insert *on*
	Bit 6	Caps Lock *on*
	Bit 5	NumLock *on.*
	Bit 4	Scroll Lock *on*
	Bit 3	Either Alt key down
	Bit 2	Either Ctrl key down
	Bit 1	Left Shift key down
	Bit 0	Right Shift key down
Time (ULONG)	Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.	

KBD_XLATESCAN (79h) - KbdDD Flags

KbdDD Flags

Defined in the Device Monitor packets. Refer to the *OS/2 Input/Output Device Driver Reference* for more information on the keyboard device driver.

KBD_XLATESCAN (79h) - Xlate Flags

Xlate Flags

Has the following:

High Byte	Bit settings are as follows:	
	Bits 8-15	Reserved. Set to 0.
Low Byte	Bit settings are as follows:	
	Bits 1-7	Reserved. Set to 0.
	Bit 0	Translation complete.

KBD_XLATESCAN (79h) - Xlate State1

Xlate State1

Identifies the state of translation across successive calls. Initially this WORD should be zero and must be reset to 0 when the caller wants a new start to translation. Notice that it can take several calls to this IOCtl to complete a character, so this field should not be touched unless a fresh start to translation is desired. Also, this field is set to 0 at the completion of translation.

KBD_XLATESCAN (79h) - Xlate State2

Xlate State2

Same as Xlate State1.

KBD_XLATESCAN (79h) - Data Packet Format

Field	Length	C Datatype
CharData Record	10 BYTES	KBDKEYINFO
KbdDD Flags	WORD	USHORT
Xlate Flags	WORD	USHORT
Xlate State1	WORD	USHORT
Xlate State2	WORD	USHORT

KBD_XLATESCAN (79h) - Returns

The error return code for this function is as follows:

13H	INVALID_PARAMETER

KBD_XLATESCAN (79h) - Remarks

This request translates a scan code in a CharData Record to an ASCII character. Optionally, a code page can be specified to use for translation. Otherwise, the code page of the active KCB is used. To translate a given scan code with a particular shift state, indicate the shift state desired with the Shift State field of the CharData Record.

KBD_XLATESCAN (79h) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_XLATESCAN (79h)
Description:
Translate Scan Code to ASCII

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_QUERYKBDHARDWAREID (7Ah) - Query Keyboard Hardwar

KBD_QUERYKBDHARDWAREID (7Ah) - Description

This function returns the Hardware ID for the currently attached keyboard.

KBD_QUERYKBDHARDWAREID (7Ah) - Parameter Packet Forma

None.

KBD_QUERYKBDHARDWAREID (7Ah) - Data Length

Data Length

On input, Data Length specifies the maximum number of returned data bytes requested by the caller. The Length value on input includes the length field's size. An input length field value of less than *2* is returned to the caller with an INVALID_PARAMETER error.

On output, Data Length always indicates the maximum number of bytes that this function can return. For OS/2 2.1, this return length size is 4 bytes.

KBD_QUERYKBDHARDWAREID (7Ah) - Hardware ID

Hardware ID

The attached Hardware ID.

KBD_QUERYKBDHARDWAREID (7Ah) - Data Packet Format

Field	Length	C Datatype
Data Length	WORD	USHORT
Hardware ID	WORD	USHORT

KBD_QUERYKBDHARDWAREID (7Ah) - Returns

Returns the Hardware ID for the currently attached keyboard.

The error return code for this function is as follows:

13H INVALID_PARAMETER

KBD_QUERYKBDHARDWAREID (7Ah) - Remarks

In the past, all keyboards could be supported by knowing the hardware family information available through KBD_GETKEYBDTYPE. The 122-key keyboard has a number of differences from the other 88/89 Key Enhanced Family keyboards. Therefore, applications performing keystroke-specific functions might need to determine which keyboard is attached.

OS/2 system-supported keyboards and their hardware generated IDs, are as follows:

Keyboard	Hardware ID
PC AT Standard Keyboard	0001H
101-Key Enhanced Keyboard	AB41H
102-Key Enhanced Keyboard	AB41H

```
 88-Key Enhanced Keyboard                          AB54H

 89-Key Enhanced Keyboard                          AB54H

122-Key Mainframe Interactive (MFI) Keyboard       AB86H
```

KBD_QUERYKBDHARDWAREID (7Ah) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_QUERYKBDHARDWAREID (7Ah)
Description:
Query Keyboard Hardware ID

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Query Keyboard Information

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Description

This function returns the current keyboard code-page support information.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Parameter Packe

None.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Data Length

Data Length
The input return data length requested or the output data structure length required, in bytes. On input, the length field value specifies the caller's return data area size. A minimum value of 2 is required so that the length value can be properly returned. An INVALID_PARM error code is returned for input length values less than 2.

On output, the remaining returned data depends on the input length field value. Length field values of 12 bytes or greater result in the full data structure being returned. For length requests less than 12, only the portion of returned data fields that can be completely returned (including ASCIIZ string terminators) is returned. The *Length* field on return always represents the byte count of returned data including string terminators.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - KbdCP

KbdCP

The active keyboard code page number.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Ctry

Ctry

The active keyboard Country code in ASCIIZ string format.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - SCtry

SCtry

The active keyboard Subcountry code in ASCIIZ string format.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Data Packet Forr

Field	Length	C Datatype
Data Length	WORD	USHORT
KbdCP	WORD	USHORT
Ctry	ASCIIZ	PSZ
SCtry	ASCIIZ	PSZ

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Returns

The error return code for this function is as follows:

8113H	INVALID_PARAMETER

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) - Remarks

This request returns the keyboard's current code-page support information. This information is particularly useful for utilities like *KEYB*. The code-page support information consists of the keyboard's active code page and the Country and Subcountry codes active for the keyboard layout. These items are the system-level support factors used by the physical Keyboard device driver for global translation.

KBD_QUERYKBDCODEPAGESUPPORT (7Bh) -

Category:
IOCTL_KEYBOARD (04h)
Function:
KBD_QUERYKBDCODEPAGESUPPORT (7Bh)
Description:
Query Keyboard Code Page Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 05h Parallel Port Control IOCtl Commands

The following is a summary of Category 05h IOCtl Commands:

Function	Description
42h	Set Frame Control (CPL, LPI)
43h	Reserved
44h	Set Infinite Retry
45h	Reserved
46h	Initialize Parallel Port
47h	Reserved
48h	Activate Font
49h	Reserved
4Bh	Reserved
4Ch	Reserved
4Dh	Set Print-Job Title
4Eh	Set Parallel Port Write Timeout Value
4Fh	Reserved
50h	Reserved
51h	Reserved
62h	Query Frame Control
63h	Reserved

64h	Query Infinite Retry
65h	Reserved
66h	Query Parallel Port Status
67h	Reserved
68h	Reserved
69h	Query Active Font
6Ah	Verify Font
6Bh	Reserved
6Ch	Reserved
6Dh	Reserved
6Eh	Query Parallel Port Write Timeout Value
6Fh	Reserved
70h	Reserved
71h	Reserved

PRT_SETFRAMECTL (42h) - Set Frame Control

PRT_SETFRAMECTL (42h) - Description

This function sets frame control.

PRT_SETFRAMECTL (42h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_SETFRAMECTL (42h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_SETFRAMECTL (42h) - Characters Per Line

Characters Per Line
Valid characters per line (CPL) value. For the SBCS code page, valid values are 80 and 132.

PRT_SETFRAMECTL (42h) - Lines per Inch

Lines per Inch
Valid lines per inch (LPI) value. For the SBCS code page, valid values are 6 and 8.

PRT_SETFRAMECTL (42h) - Data Packet Format

Field	Length	C Datatype
Characters Per Line	BYTE	BYTE
Lines per Inch	BYTE	BYTE

Related C Data Structure

The FRAME data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PRT_SETFRAMECTL (42h) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
02H	Device not ready, if monitors not registered
03H	Invalid command, if the CPL/LPI combination specified is not valid.
0AH	Write fault, if monitor registered and the DevHlp_MonWrite fails

When the printer device driver receives this IOCtl, it generates the proper data stream for the attached printer. This IOCtl is dependent on the attached printer.

If the spooler is active, the CPL/LPI setting by the MODE command does not work and returns error.

- IBM-J Printer

 The printer device driver supports following combinations of CPL and LPI in the DBCS code page if the spooler is inactive:

  ```
  [CPL] 132, 158, 176, 198.

  [LPI] 4, 5, 6, 7 (7.5).
  ```

The CPL/LPI setting used by the MODE command does not function correctly in the SBCS code page.

- U.S. Printer and ESC/P Printer

 The printer device driver supports the following combinations of CPL and LPI, if the spooler is inactive:

  ```
  [CPL] 80, 132.

  [LPI] 6, 8.
  ```

- ESC/P printer

 The CPL setting is only effective using ANK characters, but it is not effective using KANJI characters. The implementation is the same as DOS/V.

PRT_SETFRAMECTL (42h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_SETFRAMECTL (42h)
Description:
Set Frame Control

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_SETINFINITERETRY (44h) - Set Infinite Retry

PRT_SETINFINITERETRY (44h) - Description

This function sets infinite retry.

PRT_SETINFINITERETRY (44h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_SETINFINITERETRY (44h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_SETINFINITERETRY (44h) - Data

Data

The data is defined as:

0	Disable infinite retry
1	Enable infinite retry

PRT_SETINFINITERETRY (44h) - Data Packet Format

Field	Length	C Datatype
Data	BYTE	UCHAR

PRT_SETINFINITERETRY (44h) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
03H	Invalid command, if data not 0 or 1

PRT_SETINFINITERETRY (44h) - Remarks

If a monitor is registered for the physical parallel port device driver, infinite retry is always enabled. When a monitor is registered, a disable request returns a successful return code and the function is not performed.

PRT_SETINFINITERETRY (44h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_SETINFINITERETRY (44h)
Description:

Set Infinite Retry

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PRT_INITPRINTER (46h) - Initialize Parallel Port Device

PRT_INITPRINTER (46h) - Description

This function initializes a parallel port attached device.

PRT_INITPRINTER (46h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_INITPRINTER (46h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_INITPRINTER (46h) - Data Packet Format

Field	Length	C Datatype
Set to zero	BYTE	UCHAR

PRT_INITPRINTER (46h) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
0AH	Write fault, if monitor registered and the DevHlp_MonWrite fails

PRT_INITPRINTER (46h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_INITPRINTER (46h)
Description:
Initialize Parallel Port Device

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_ACTIVATEFONT (48h) - Activate Font

PRT_ACTIVATEFONT (48h) - Description

This function activates a font.

PRT_ACTIVATEFONT (48h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_ACTIVATEFONT (48h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_ACTIVATEFONT (48h) - Code Page

Code Page

The value of the code page to make active.

0000H	If the Code Page value and Font ID are specified as zero, set printer to hardware default code page and font.
0001H-FFFFH	Valid code page numbers.

PRT_ACTIVATEFONT (48h) - Font ID

Font ID

The ID value of the font to make currently active.

0000H	If the Code Page value and Font ID are specified as zero, set printer to hardware default code page and font. If Font ID is zero and the code page is a valid nonzero, then any font within the specified code page is acceptable.
0001H-FFFFH	Valid Font ID numbers; font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer.

PRT_ACTIVATEFONT (48h) - Data Packet Format

Field	Length	C Datatype
Code Page	WORD	USHORT
Font ID	WORD	USHORT

PRT_ACTIVATEFONT (48h) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
0AH	Write fault, if monitor registered and the DevHlp_MonWrite fails
02H	Code page is not available
03H	No code page function because spooler not started
04H	Font ID is not available (verify)
09H	Error caused by switcher error, not by input parameters
0AH	Error caused by invalid printer name as input

0DH	Received code page request when code page switcher not initialized
0FH	SFN table full: cannot activate another entry
13H	DASD error reading font file
15H	DASD error reading font file definition block
17H	DASD error while writing to temporary spool file
18H	Disk full error while writing to temporary spool file
19H	Spool file handle was bad

PRT_ACTIVATEFONT (48h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_ACTIVATEFONT (48h)
Description:
Activate Font

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_SETPRINTJOBTITLE (4Dh) - Set Print-Job Title

PRT_SETPRINTJOBTITLE (4Dh) - Description

This function informs the base parallel port device driver (and ultimately to the spooler) of the file to be printed. The print job title appears in the print object. If the spooler is not started, the request may not perform a function.

PRT_SETPRINTJOBTITLE (4Dh) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_SETPRINTJOBTITLE (4Dh) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_SETPRINTJOBTITLE (4Dh) - Length

Length
The length of the buffer containing the print-job title. This includes the terminating null character but excludes the *Length* field.

PRT_SETPRINTJOBTITLE (4Dh) - Address of ASCIIZ Job Title

Address of ASCIIZ Job Title
The 16:16 address of the ASCIIZ string containing the print-job title. A null character terminates the ASCIIZ string.

PRT_SETPRINTJOBTITLE (4Dh) - Data Packet Format

Field	Length	C Datatype
Length	WORD	USHORT
Address of ASCIIZ Job Title	DWORD	ULONG

PRT_SETPRINTJOBTITLE (4Dh) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
13H	Invalid parameter, if the address of data packet is invalid or the address of the ASCIIZ string is invalid

PRT_SETPRINTJOBTITLE (4Dh) - Remarks

If the ASCII string has more than 126 bytes, the physical parallel port device driver truncates it to the first 125 bytes plus 1 byte for the null character.

PRT_SETPRINTJOBTITLE (4Dh) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_SETPRINTJOBTITLE (4Dh)
Description:
Set Print-Job Title

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PRT_SETIRQTIMEOUT (4Eh) - Set Parallel Port Write Timeout Val

PRT_SETIRQTIMEOUT (4Eh) - Description

This function sets the parallel port write timeout value.

PRT_SETIRQTIMEOUT (4Eh) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_SETIRQTIMEOUT (4Eh) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_SETIRQTIMEOUT (4Eh) - Timeout Value in Seconds

Timeout Value in Seconds
The length of time, in seconds, that the physical parallel port device driver waits for a write hardware interrupt to occur.

PRT_SETIRQTIMEOUT (4Eh) - Data Packet Format

Field	Length	C Datatype
Timeout Value in Seconds	WORD	USHORT

PRT_SETIRQTIMEOUT (4Eh) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
13H	Invalid parameter, if the data-packet address is invalid

PRT_SETIRQTIMEOUT (4Eh) - Remarks

This IOCtl informs the physical parallel port device driver of the length of time, in seconds, to wait for a hardware interrupt before a write request is canceled. Valid timeout values range between 0 and 65535 seconds.

PRT_SETIRQTIMEOUT (4Eh) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_SETIRQTIMEOUT (4Eh)
Description:
Set Parallel Port Write Timeout Value

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PRT_GETFRAMECTL (62h) - Query Frame Control

PRT_GETFRAMECTL (62h) - Description

This function returns frame control.

PRT_GETFRAMECTL (62h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_GETFRAMECTL (62h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_GETFRAMECTL (62h) - Characters per Line

Characters per Line
On return, field is set to the characters per line. For the SBCS code page, valid values are 80 and 132.

PRT_GETFRAMECTL (62h) - Lines per Inch

Lines per Inch
On return, field is set to the lines per inch. For the SBCS code page, valid values are 6 and 8.

PRT_GETFRAMECTL (62h) - Data Packet Format

Field	Length	C Datatype
Characters per Line	BYTE	BYTE
Lines per Inch	BYTE	BYTE

Related C Data Structure

The FRAME data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PRT_GETFRAMECTL (62h) - Returns

The error return code for this function is as follows:

00H Completed successfully

This IOCtl is dependent on the attached printer. If the spooler is active, the CPL/LPI setting by the MODE command does not work and returns error.

- IBM-J Printer

 The printer device driver supports following combinations of CPL and LPI in the DBCS code page if the spooler is inactive:

  ```
  [CPL] 132, 158, 176, 198.

  [LPI] 4, 5, 6, 7 (7.5).
  ```

 The CPL/LPI setting used by the MODE command does not function correctly in the SBCS code page.

- U.S. Printer and ESC/P Printer

 The printer device driver supports the following combinations of CPL and LPI, if the spooler is inactive:

  ```
  [CPL] 80, 132.

  [LPI] 6, 8.
  ```

- ESC/P printer

 The CPL setting is only effective using ANK characters, but it is not effective using KANJI characters. The implementation is the same as DOS/V.

PRT_GETFRAMECTL (62h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_GETFRAMECTL (62h)
Description:
Query Frame Control

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_GETINFINITERETRY (64h) - Query Infinite Retry

PRT_GETINFINITERETRY (64h) - Description

Returns the infinite retry.

PRT_GETINFINITERETRY (64h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_GETINFINITERETRY (64h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_GETINFINITERETRY (64h) - Data

Data
On return, Data byte is set to:

0	Infinite retry is disabled.
1	Infinite retry is enabled.

PRT_GETINFINITERETRY (64h) - Data Packet Format

Field	Length	C Datatype
Data	BYTE	UCHAR

PRT_GETINFINITERETRY (64h) - Returns

The error return code for this function is as follows:

00H Completed successfully.

When a printer monitor is installed, infinite retry is enabled. In this case, a disable request will return a successful return code, however the function will not be performed.

PRT_GETINFINITERETRY (64h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_GETINFINITERETRY (64h)
Description:
Query Infinite Retry

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_GETPRINTERSTATUS (66h) - Query Parallel Port Status

PRT_GETPRINTERSTATUS (66h) - Description

Returns the parallel port status.

PRT_GETPRINTERSTATUS (66h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_GETPRINTERSTATUS (66h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_GETPRINTERSTATUS (66h) - Data

Data

On return, Data byte is set to:

Bit 7	1 = Not Busy
Bit 6	1 = Acknowledge
Bit 5	1 = Out of Paper
Bit 4	1 = Selected
Bit 3	1 = I/O Error
Bit 2	Unused
Bit 1	Unused
Bit 0	1 = Timeout

PRT_GETPRINTERSTATUS (66h) - Data Packet Format

Field	Length	C Datatype
Data	BYTE	UCHAR

PRT_GETPRINTERSTATUS (66h) - Returns

The error return code for this function is as follows:

00H Completed successfully

PRT_GETPRINTERSTATUS (66h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_GETPRINTERSTATUS (66h)
Description:
Query Parallel Port Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_QUERYACTIVEFONT (69h) - Query Active Font

PRT_QUERYACTIVEFONT (69h) - Description

Queries an active font.

PRT_QUERYACTIVEFONT (69h) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_QUERYACTIVEFONT (69h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_QUERYACTIVEFONT (69h) - Code Page

Code Page
On return, is set to currently active code page.

0000H	If the Code Page value and Font ID are returned as zero, the printer is set to the hardware default code page and font.
0001H-FFFFH	Valid code page numbers.

PRT_QUERYACTIVEFONT (69h) - Font ID

Font ID
On return, is the ID value of the font that is currently active.

0000H	If the Code Page value and Font ID are specified as zero, the printer is set to the hardware default code page and font. If Font ID is zero and code page is nonzero, no error is returned if any Font ID is available for the specified code page.
0001H-FFFFH	Valid Font ID numbers; font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label, and are also entered in the DEVINFO statement for the printer.

PRT_QUERYACTIVEFONT (69h) - Data Packet Format

Field	Length	C Datatype
Code Page	WORD	USHORT
Font ID	WORD	USHORT

PRT_QUERYACTIVEFONT (69h) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
03H	No code page function because spooler not started
09H	Error caused by switcher error, not by input parameters
0AH	Error caused by invalid printer name as input
0DH	Received code page request when code page switcher not initialized
10H	Received request for SFN, not in SFN table

PRT_QUERYACTIVEFONT (69h) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_QUERYACTIVEFONT (69h)
Description:
Query Active Font

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_VERIFYFONT (6Ah) - Verify Font

PRT_VERIFYFONT (6Ah) - Description

Verifies that a particular code page and font are available for the specified printer.

PRT_VERIFYFONT (6Ah) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_VERIFYFONT (6Ah) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_VERIFYFONT (6Ah) - Code Page

Code Page
The Code Page number to validate. Values can be 0 - 65535.

PRT_VERIFYFONT (6Ah) - Font ID

Font ID
The Font ID value to validate. Values can be 0 - 65535. The Font ID is contained in the font file for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label, and are entered in the DEVINFO statement for the printer.

Note: A value of *0* for both the Code Page number and Font ID indicates the default hardware code page and font. This combination is always valid.

PRT_VERIFYFONT (6Ah) - Data Packet Format

Field	Length	C Datatype
Code Page	WORD	USHORT
Font ID	WORD	USHORT

PRT_VERIFYFONT (6Ah) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
02H	Code page is not available
03H	No code page function because spooler not started
04H	Font ID is not available (verify)
0AH	Error caused by invalid printer name as input
0DH	Received code page request when code page switcher not initialized

PRT_VERIFYFONT (6Ah) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_VERIFYFONT (6Ah)
Description:
Verify Font

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

PRT_QUERYIRQTIMEOUT (6Eh) - Query Parallel Port Write Timec

PRT_QUERYIRQTIMEOUT (6Eh) - Description

Returns a parallel port write timeout value.

PRT_QUERYIRQTIMEOUT (6Eh) - Command Information

Command Information
Reserved. Must be set to 0.

PRT_QUERYIRQTIMEOUT (6Eh) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PRT_QUERYIRQTIMEOUT (6Eh) - Timeout Value in Seconds

Timeout Value in Seconds
The length of time, in seconds, that the physical parallel port device driver waits for a write hardware interrupt to occur.

PRT_QUERYIRQTIMEOUT (6Eh) - Data Packet Format

Field	Length	C Datatype
Timeout Value in Seconds	WORD	USHORT

PRT_QUERYIRQTIMEOUT (6Eh) - Returns

The error return codes for this function are as follows:

00H	Completed successfully
13H	Invalid parameter, if the data-packet address is invalid

PRT_QUERYIRQTIMEOUT (6Eh) - Remarks

Valid timeout values range between 0 - 65535 seconds.

PRT_QUERYIRQTIMEOUT (6Eh) -

Category:
IOCTL_PRINTER (05h)
Function:
PRT_QUERYIRQTIMEOUT (6Eh)
Description:
Query Parallel Port Write Timeout Value

Select an item:
Description

Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 07h Mouse Control IOCtl Commands

The following is a summary of Category 07h IOCtl Commands:

Function	Description
50h	Reserved
51h	Notification of Display Mode Change
52h	Reserved
53h	Reassign Current Mouse Scaling Factors
54h	Assign New Mouse Event Mask
55h	Reassign Mouse Threshold Values
56h	Set Pointer Shape
57h	Unmark Collision Area
58h	Mark Collision Area
59h	Specify/Replace Pointer Screen Position
5Ah	Set OS/2 Mode Pointer Draw Device Driver Address
5Bh	Reserved
5Ch	Set Current Physical Mouse Device Driver Status Flags
5Dh	Notification of Mode Switch Completion
60h	Query Number of Mouse Buttons Supported
61h	Query Mouse Device Motion Sensitivity
62h	Query Current Physical Mouse Device Driver Status Flags
63h	Read Mouse Event Queue
64h	Query Current Event Queue Status
65h	Query Current Mouse Event Mask
66h	Query Current Mouse Scaling Factors
67h	Query Current Pointer Screen Position
68h	Query Current Pointer Shape
69h	Query Mouse Threshold Values
6Ah	Query Physical Mouse Device Driver Level/Version Number
6Bh	Query Pointing Device ID

MOU_UPDATEDISPLAYMODE (51h) - Notification of Display Mode

MOU_UPDATEDISPLAYMODE (51h) - Description

This function performs a notification of display mode change.

MOU_UPDATEDISPLAYMODE (51h) - Length

Length

An input parameter to VioSetMode. Length specifies the length of the data structure in bytes including Length itself. The minimum structure size required is 3 bytes; the maximum structure size required is 34 bytes. OS/2 2.1 sets the first mode (in the list of modes supported by this display configuration) with a data structure matching the mode data specified.

MOU_UPDATEDISPLAYMODE (51h) - Type

Type

A bit mask that contains specifications for the mode being set. The definitions of the bits follow:

```
xxxxxxxb b = 0  Monochrome compatible mode
         b = 1  Other
xxxxxxbx b = 0  Text mode
         b = 1  Graphics mode
xxxxxbxx b = 0  Enable color burst
         b = 1  Disable color burst
xxxxbxxx b = 0  VGA-compatible modes 0 - 13H
         b = 1  Native mode
bbbbxxxx b = Reserved, must be 0
```

MOU_UPDATEDISPLAYMODE (51h) - Color

Color

Defines the number of colors as a power of 2. This is equivalent to the number of color bits, which define the color. For example:

```
Color = 1  Specifies 2 colors
Color = 2  Specifies 4 colors
Color = 4  Specifies 16 colors
Color = 8  Specifies 256 colors
Color = 0  Should be specified for monochrome modes 7, 7+, and F
```

MOU_UPDATEDISPLAYMODE (51h) - Text Columns

Text Columns
Number of text columns.

MOU_UPDATEDISPLAYMODE (51h) - Text Rows

Text Rows
Number of text rows. Twenty-five rows are supported for the color graphics adapter. Twenty-five and forty-three rows are supported for the VGA adapter. Twenty-five and fifty rows are supported for the IBM Personal System/2* Display Adapter.

MOU_UPDATEDISPLAYMODE (51h) - Horizontal Resolution

Horizontal Resolution
The number of pel columns.

MOU_UPDATEDISPLAYMODE (51h) - Vertical Resolution

Vertical Resolution
The number of pel rows.

MOU_UPDATEDISPLAYMODE (51h) - Attribute Format

Attribute Format
Identifies the format of the attributes.

MOU_UPDATEDISPLAYMODE (51h) - Number of Attributes

Number of Attributes
Identifies the number of attributes in a character cell.

MOU_UPDATEDISPLAYMODE (51h) - Buffer Address

Buffer Address
The 32-bit physical address of the physical display buffer for this mode (returned value).

MOU_UPDATEDISPLAYMODE (51h) - Buffer Length

Buffer Length
The length of the physical display buffer for this mode (returned value).

MOU_UPDATEDISPLAYMODE (51h) - Full Buffer Size

Full Buffer Size
The size of the buffer required for a full save of the physical display buffer for this mode (returned value).

MOU_UPDATEDISPLAYMODE (51h) - Partial Buffer Size

Partial Buffer Size
The size of the buffer required for a partial (pop-up) save of the physical display buffer for this mode (returned value).

MOU_UPDATEDISPLAYMODE (51h) - Address of Extended Data A

Address of Extended Data Area
The Far-16 address to an *extended-mode* data structure, or zero, if none. The format of the extended-mode data structure is determined by the physical device driver and is unknown to the operating system.

MOU_UPDATEDISPLAYMODE (51h) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains the Mode Data Definition record, which has the following format:

Field	Length	C Datatype
Length	WORD	USHORT
Type	BYTE	UCHAR
Color	BYTE	UCHAR
Text Columns	WORD	USHORT
Text Rows	WORD	USHORT

Horizontal Resolution	WORD	USHORT
Vertical Resolution	WORD	USHORT
Attribute Format	BYTE	UCHAR
Number of Attributes	BYTE	UCHAR
Buffer Address	DWORD	ULONG
Buffer Length	DWORD	ULONG
Full Buffer Size	DWORD	ULONG
Partial Buffer Size	DWORD	ULONG
Address of Extended Data Area	DWORD	ULONG

MOU_UPDATEDISPLAYMODE (51h) - Length

Length

An input parameter to VioGetConfig. Length specifies the length of the data structure in bytes including Length itself. The maximum size structure required is determined by issuing VioGetConfig with Length set to 2. When Length is set to 2 on input, VioGetConfig returns the maximum size structure required in the Length field on output. When Length is not equal to 2 on input, the Length field is unmodified on output.

MOU_UPDATEDISPLAYMODE (51h) - Adapter Type

Adapter Type

The display adapter type, with values set as follows:

0 = Monochrome-compatible
1 = Color graphics adapter
2 = Enhanced graphics adapter
3 = VGA or PS/2* display adapter
4-6 = Reserved
7 = PS/2 Display Adapter 8514/A
8 = PS/2 Image Adapter/A
9 = PS/2 XGA Display Adapter
10 = PS/2 SVGA Display Adapter

Note: Values ranging from 0-4095 are reserved for IBM.

MOU_UPDATEDISPLAYMODE (51h) - Display Type

Display Type

Type of display/monitor with values indicating the following:

0 = Monochrome display
1 = Color display
2 = Enhanced color display
3 = 8503 monochrome display
4 = 8512 or 8513 color display
5-8 = Reserved

9 = 8514 color display
10 = Plasma display panel
11 = Monochrome Displays 8507 and 8604
12 = PS/2 Color Display 8515
13 = Reserved

Note: Values ranging from 0-4095 are reserved for IBM.

MOU_UPDATEDISPLAYMODE (51h) - Memory

Memory
The amount of memory on the adapter (in bytes) returned as a 32-bit value.

MOU_UPDATEDISPLAYMODE (51h) - Configuration Number

Configuration Number
The number of the display configuration to which this data corresponds.

MOU_UPDATEDISPLAYMODE (51h) - Device Driver Version Numb

Device Driver Version Number
The video device driver version number corresponding to this device driver.

MOU_UPDATEDISPLAYMODE (51h) - Flag Bits

Flag Bits
Defined as follows:

0001H	Power-up display configuration
0002H	VGA pass-through.

MOU_UPDATEDISPLAYMODE (51h) - Hardware State Buffer Size

Hardware State Buffer Size
The size of the buffer required by the video device driver to save the full hardware state excluding the physical display buffer.

MOU_UPDATEDISPLAYMODE (51h) - Maximum Buffer Size - Full

Maximum Buffer Size - Full Save
The maximum size buffer required by the video device driver to save the full physical display buffer.

MOU_UPDATEDISPLAYMODE (51h) - Maximum Buffer Size - Part

Maximum Buffer Size - Partial Save
The maximum size buffer required by the video device driver to save the portions of the physical display buffer that are overlaid by a pop-up.

MOU_UPDATEDISPLAYMODE (51h) - Offset to Mode Data

Offset to Mode Data
The offset within the configuration data structure to the beginning of the mode data.

MOU_UPDATEDISPLAYMODE (51h) - Data Packet Format

The Data Packet is a location in application storage that contains the configuration data for the display on which the mode is being set. The configuration data has the following format:

Field	Length	C Datatype
Length	WORD	USHORT
Adapter Type	WORD	USHORT
Display Type	WORD	USHORT
Memory	DWORD	USHORT
Configuration Number	WORD	USHORT
Device Driver Version Number	WORD	USHORT
Flag Bits	WORD	USHORT
Hardware State Buffer Size	DWORD	ULONG
Maximum Buffer Size - Full Save	DWORD	ULONG
Maximum Buffer Size - Partial Save	DWORD	ULONG
Offset to Mode Data	WORD	USHORT

MOU_UPDATEDISPLAYMODE (51h) - Remarks

This function notifies the physical mouse device driver of a new display mode and display configuration. When the video subsystem or registered video subsystem sets or resets the display mode, it must synchronize the physical mouse device driver pointer update routines (by providing this notification record to the physical mouse device driver) before switching display modes.

MOU_UPDATEDISPLAYMODE (51h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_UPDATEDISPLAYMODE (51h)
Description:
Notification of Display Mode Change

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

MOU_SETSCALEFACTORS (53h) - Reassign Current Mouse Scali

MOU_SETSCALEFACTORS (53h) - Description

This function reassigns the current mouse scaling factors.

MOU_SETSCALEFACTORS (53h) - Row Data

Row Data
Row-coordinate scaling factor.

MOU_SETSCALEFACTORS (53h) - Column Data

Column Data
The column-coordinate scaling factor. The scaling factor values are positive integers in the range of:

```
0 < value <= (32K-1)
```

MOU_SETSCALEFACTORS (53h) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Row Data	WORD	USHORT
Column Data	WORD	USHORT

MOU_SETSCALEFACTORS (53h) - Data Packet Format

None.

MOU_SETSCALEFACTORS (53h) - Returns

None.

MOU_SETSCALEFACTORS (53h) - Remarks

This function reassigns the current pointing device scaling factors. *Scaling factors* are ratio values that determine how much relative movement is necessary before the physical Mouse device driver reports a mouse event. These ratios specify the number of mickeys per 8 pels. The default ratio values are:

Vertical/Row ratio - 16 mickeys per 8 pels
Horizontal/Column ratio - 8 mickeys per 8 pels

MOU_SETSCALEFACTORS (53h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_SETSCALEFACTORS (53h)
Description:
Reassign Current Mouse Scaling Factors

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_SETEVENTMASK (54h) - Assign New Mouse Event Mask

MOU_SETEVENTMASK (54h) - Description

This function assigns a new mouse event mask.

MOU_SETEVENTMASK (54h) - Event Mask

Event Mask

Has the following bits:

Bits 7-15	Reserved. Set to 0.
Bit 6	Set, if Button 3 is down
Bit 5	Set, if motion with Button 3 down
Bit 4	Set, if Button 2 is down
Bit 3	Set, if motion with Button 2 down
Bit 2	Set, if Button 1 is down
Bit 1	Set, if motion with Button 1 down
Bit 0	Set, if all mouse motion, no buttons down

A set bit has a value of 1.

MOU_SETEVENTMASK (54h) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Event Mask	WORD	USHORT

MOU_SETEVENTMASK (54h) - Data Packet Format

None.

MOU_SETEVENTMASK (54h) - Returns

None.

MOU_SETEVENTMASK (54h) - Remarks

The physical mouse device driver gets the new mask for enabled/disabled mouse device events from the caller's Parameter Packet. This mask determines which events are to be queued (that is, to be read by calling MOU_READEVENTQUE).

MOU_SETEVENTMASK (54h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_SETEVENTMASK (54h)
Description:
Assign New Mouse Event Mask

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_REASSIGNTHRESHOLDVALUES (55h) - Reassign Mouse T

MOU_REASSIGNTHRESHOLDVALUES (55h) - Description

This function sets the mouse acceleration parameters.

MOU_REASSIGNTHRESHOLDVALUES (55h) - Length

Length
This should be initialized with the size of the parameter packet in bytes.

MOU_REASSIGNTHRESHOLDVALUES (55h) - 1st Movement Lev

1st Movement Level
This value determines when the first level multiplier is applied.

MOU_REASSIGNTHRESHOLDVALUES (55h) - 1st Level Multiplier

1st Level Multiplier
This value is multiplied with the actual number of units moved to produce acceleration.

MOU_REASSIGNTHRESHOLDVALUES (55h) - 2nd Movement Lev

2nd Movement Level
This values determines when the second level multiplier is applied.

MOU_REASSIGNTHRESHOLDVALUES (55h) - 2nd Level Multiplie

2nd Level Multiplier
This value is multiplied with the actual number of units moved to produce acceleration.

MOU_REASSIGNTHRESHOLDVALUES (55h) - Parameter Packet

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Length	WORD	USHORT
1st Movement Level	WORD	USHORT
1st Level Multiplier	WORD	USHORT
2nd Movement Level	WORD	USHORT
2nd Level Multiplier	WORD	USHORT

MOU_REASSIGNTHRESHOLDVALUES (55h) - Data Packet Forma

None.

MOU_REASSIGNTHRESHOLDVALUES (55h) - Returns

None.

MOU_REASSIGNTHRESHOLDVALUES (55h) - Remarks

For each mouse event, the count of physical units moved is compared against the 1st and 2nd level thresholds. If the count is greater than the 2nd level threshold, the 2nd level multiplier will be used; if the count is greater than the 1st level but less than the 2nd level, the 1st level multiplier will be used. If the count is less than the 1st level threshold, no multiplier will be used. These calculations are done separately for the X- and Y-axis.

MOU_REASSIGNTHRESHOLDVALUES (55h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_REASSIGNTHRESHOLDVALUES (55h)
Description:
Reassign Mouse Threshold Values

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_SETPTRSHAPE (56h) - Set Pointer Shape

MOU_SETPTRSHAPE (56h) - Description

This function sets the pointer shape.

MOU_SETPTRSHAPE (56h) - Buffer Length

Buffer Length
The length of the pointer image buffer.

MOU_SETPTRSHAPE (56h) - Columns

Columns
The width, in columns, of the pointer image.

MOU_SETPTRSHAPE (56h) - Rows

Rows
The height, in rows, of the pointer image.

MOU_SETPTRSHAPE (56h) - Column Hot Spot

Column Hot Spot
The column offset (within the pointer image) to the hot spot.

MOU_SETPTRSHAPE (56h) - Row Hot Spot

Row Hot Spot
The row offset (within the pointer image) to the hot spot.

MOU_SETPTRSHAPE (56h) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following Pointer Definition record. The Pointer Definition record has the following format:

Field	Length	C Datatype
Buffer Length	WORD	USHORT
Columns	WORD	USHORT
Rows	WORD	USHORT
Column Hot Spot	WORD	USHORT
Row Hot Spot	WORD	USHORT

MOU_SETPTRSHAPE (56h) - Data Packet Format

The Data Packet is a location in application storage that contains the Pointer Image buffer. The Pointer Image buffer contains a new pointer shape defined by the user for the physical Mouse device driver. The format of this buffer is dependent on the display mode. The buffer always consists of the and pointer image data followed by the XOR pointer image data. The buffer always describes only one display plane.

MOU_SETPTRSHAPE (56h) - Returns

None.

MOU_SETPTRSHAPE (56h) - Remarks

In the Parameter Packet, the caller specifies a 1-WORD height value (Rows) for the pointer shape, and a 1-WORD width value (Columns) for the pointer shape. For text-mode and mono-mode applications, these values must be equal to 1. For graphics-mode applications, these values must be greater than or equal to 1.

The Length, in bytes, of the pointer shape varies, depending on the display mode:

```
Mono & Text
Buffer length = (height in characters) * (width in characters) * 2 * 2
              = 1 * 1 * 2 * 2
                Notice that for text mode, height and width must be 1, so length is always 4.

Graphics
Buffer length = (height in pels) * (width in pels) * (bits per pel) * 2 / 8
                Notice that width must be a multiple of 8.

Modes 4 & 5 (320 x 200)
Buffer length = (height) * (width) * 2 * 2 / 8

Mode 6 (640 x 200)
Buffer length = (height) * (width) * 1 * 2 / 8
                Notice that length calculations produce byte boundary buffer sizes.
```

All of the Pointer Definition record fields and the Pointer Shape buffer are validated using the session's mode table values. The parameter values must be the same orientation as the current session display mode:

- Graphics = pel values
- Text = character values

MOU_SETPTRSHAPE (56h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_SETPTRSHAPE (56h)
Description:
Set Pointer Shape

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_UNMARKCOLLISIONAREA (57h) - Unmark Collision Area

MOU_UNMARKCOLLISIONAREA (57h) - Description

This function frees the mouse to draw the pointer anywhere on the screen.

MOU_UNMARKCOLLISIONAREA (57h) - Parameter Packet Forma

None.

MOU_UNMARKCOLLISIONAREA (57h) - Data Packet Format

None.

MOU_UNMARKCOLLISIONAREA (57h) - Returns

None.

MOU_UNMARKCOLLISIONAREA (57h) - Remarks

A *collision area* is a restricted area on the screen that cannot be overwritten by pointer drawing. See MOU_MARKCOLLISIONAREA.

Function 57h frees the current collision area for a session so that the pointer can be drawn anywhere on the screen. If a collision area was declared for the session, the collision area is freed and the pointer position is checked. If the pointer was in the freed area, it is drawn. If the pointer was not in the freed area, no pointer manipulations take place.

MOU_UNMARKCOLLISIONAREA (57h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_UNMARKCOLLISIONAREA (57h)
Description:
Unmark Collision Area

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_MARKCOLLISIONAREA (58h) - Mark Collision Area

MOU_MARKCOLLISIONAREA (58h) - Description

This function restricts the mouse from pointer drawing in specified areas on the screen.

MOU_MARKCOLLISIONAREA (58h) - Left Row Position

Left Row Position
Vertical starting point of the collision area. Valid values are from 0 through vertical resolution -1.

MOU_MARKCOLLISIONAREA (58h) - Left Column Position

Left Column Position
Horizontal starting point of the collision area. Valid values are from 0 through horizontal resolution -1.

MOU_MARKCOLLISIONAREA (58h) - Right Row Position

Right Row Position
Vertical ending point of the collision area. Valid values are from 0 through vertical resolution -1.

MOU_MARKCOLLISIONAREA (58h) - Right Column Row Position

Right Column Row Position
Horizontal ending point of the collision area. Valid values are from 0 through horizontal resolution -1.

MOU_MARKCOLLISIONAREA (58h) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains a data structure that specifies a restricted area on the screen (collision area). The format of this data structure follows:

Field	Length	C Datatype
Left Row Position	WORD	SHORT
Left Column Position	WORD	SHORT
Right Row Position	WORD	SHORT
Right Column Row Position	WORD	SHORT

MOU_MARKCOLLISIONAREA (58h) - Data Packet Format

None.

MOU_MARKCOLLISIONAREA (58h) - Returns

None.

MOU_MARKCOLLISIONAREA (58h) - Remarks

A *collision area* is a restricted area on the screen that cannot be overwritten by pointer drawing. Values contained in the data structure defining a collision area must be specified in either character or pel values, depending on the current display mode. If the collision area is defined as the entire screen, pointer drawing is disabled for the session.

The pointer is not drawn in this area until a different area is specified through another call to this function or until MOU_UNMARKCOLLISIONAREA is called.

MOU_MARKCOLLISIONAREA (58h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_MARKCOLLISIONAREA (58h)
Description:
Mark Collision Area

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_SETPTRPOS (59h) - Specify/Replace Pointer Screen Positio

MOU_SETPTRPOS (59h) - Description

This function specifies and replaces the pointer position.

MOU_SETPTRPOS (59h) - Row Position

Row Position
The new row-coordinate pointer screen position.

MOU_SETPTRPOS (59h) - Column Position

Column Position
The new column-coordinate pointer screen position.

MOU_SETPTRPOS (59h) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Row Position	WORD	USHORT
Column Position	WORD	USHORT

MOU_SETPTRPOS (59h) - Data Packet Format

None.

MOU_SETPTRPOS (59h) - Returns

None.

MOU_SETPTRPOS (59h) - Remarks

The coordinate values are dependent on the display mode. Pel values must be used if the display is in graphics mode. Character position values must be used if the display is in text mode.

This function does not override MOU_UNMARKCOLLISIONAREA or MOU_MARKCOLLISIONAREA as it has no effect on current restricted areas of the screen. If the pointer is directed into a restricted area (collision area), it remains invisible until moved out of the area or until the restrictions are removed.

MOU_SETPTRPOS (59h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_SETPTRPOS (59h)
Description:
Specify/Replace Pointer Screen Position

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_SETPROTDRAWADDRESS (5Ah) - Set OS/2 Mode Pointer D Address

MOU_SETPROTDRAWADDRESS (5Ah) - Description

This function specifies the address of the physical Pointer Draw device driver.

MOU_SETPROTDRAWADDRESS (5Ah) - Pointer Entry

Pointer Entry
Contains two 1-WORD fields whose format is as follows:

WORD 0	Pointer Draw device driver entry point offset
WORD 1	Pointer Draw device driver entry point selector.

MOU_SETPROTDRAWADDRESS (5Ah) - Pointer DS

Pointer DS

Contains two 1-WORD fields whose format is as follows:

WORD 0	Pointer Draw device driver data segment selector
WORD 1	Reserved=0.

MOU_SETPROTDRAWADDRESS (5Ah) - Parameter Packet Forma

The Parameter Packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Pointer Entry	DWORD	ULONG
Pointer DS	DWORD	ULONG

MOU_SETPROTDRAWADDRESS (5Ah) - Length of Data

Length of Data

The length of the data structure is equal to 6 bytes.

MOU_SETPROTDRAWADDRESS (5Ah) - Display Configuration Nu

Display Configuration Number

The display configuration number on which the pointer draw routine should draw.

MOU_SETPROTDRAWADDRESS (5Ah) - Caller

Caller

Specifies whether this call is made for an application or the Base Video Subsystem (BVS), where:

0	Application
1	BVS

MOU_SETPROTDRAWADDRESS (5Ah) - Data Packet Format

The Data Packet is a location in application storage that contains the following data:

Field	Length	C Datatype

Length of Data	WORD	USHORT
Display Configuration Number	WORD	USHORT
Caller	WORD	USHORT

Related C Data Structure

The PTRDRAWDATA data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

MOU_SETPROTDRAWADDRESS (5Ah) - Remarks

This IOCtl is for OS/2 mode only. The function passes to the physical mouse device driver the selector:offset address of the entry point of the session's pointer draw routine for OS/2-mode display support. The pointer image draw routine is an installed *pseudo* character device driver. The mouse router/handler must:

- Open the physical Pointer Draw device driver
- Query the physical Pointer Draw device driver for the address of its entry point
- Pass the resulting address of the pointer draw entry point to the physical mouse device driver using the IOCtl described above

The physical mouse device driver issues a Far-16 call to the physical Pointer Draw device driver when a mouse interrupt occurs that requires action concerning the pointer image.

In addition, the physical mouse device driver can call the pointer draw routine as a result of some action on the part of the application, such as:

- MouDrawPtr
- MouRemovePtr
- MouSetPtrPos
- MouSetPtrShape
- MouGetPtrShape
- MouSetDevStatus

MOU_SETPROTDRAWADDRESS (5Ah) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_SETPROTDRAWADDRESS (5Ah)
Description:
Set OS/2 Mode Pointer Draw Device Driver Address

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

MOU_SETMOUSTATUS (5Ch) - Set Physical Mouse Device Driver

MOU_SETMOUSTATUS (5Ch) - Description

This function sets a subset of the current physical mouse device driver Status flags.

MOU_SETMOUSTATUS (5Ch) - Status Flag

Status Flag

Has the following bit-level definitions:

High Byte	Bit settings are as follows:	
	Bits 7-2	Reserved. Set to 0.
	Bit 1	Set, if mouse data returned in mickeys, not display units
	Bit 0	Set, if the interrupt level pointer draw routine is not called.
Low Byte	Bit settings are as follows:	
	Bits 7-0	Reserved. Set to 0.

A set bit has a value of 1.

MOU_SETMOUSTATUS (5Ch) - Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Status Flag	WORD	USHORT

MOU_SETMOUSTATUS (5Ch) - Data Packet Format

None.

MOU_SETMOUSTATUS (5Ch) - Returns

None.

MOU_SETMOUSTATUS (5Ch) - Remarks

This function is the complement to MOU_GETMOUSTATUS. Only the high byte of the 1-WORD physical mouse device driver Status Flag can be set. If the Data Format flag is altered, the monitor chain and event queue are flushed for the calling session of this IOCtl.

It is useful to disable interrupt-time Pointer Draw device driver activity when an application is going to use an unsupported display mode for which it intends to provide the pointer image drawing support. If this is desired, the application must set this mouse state prior to switching the display mode.

MOU_SETMOUSTATUS (5Ch) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_SETMOUSTATUS (5Ch)
Description:
Set Physical Mouse Device Driver Status Flags

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_DISPLAYMODECHANGE (5Dh) - Notification of Mode Switch

MOU_DISPLAYMODECHANGE (5Dh) - Description

This function informs the mouse that a display mode or configuration switch has been completed, and that drawing operations can resume.

MOU_DISPLAYMODECHANGE (5Dh) - Parameter Packet Format

Field	Length	C Datatype
Dummy Far-16 Pointer (such as a doubleword of zeroes)	DWORD	ULONG

MOU_DISPLAYMODECHANGE (5Dh) - Data Packet Format

Field	Length	C Datatype
Dummy Far-16 Pointer (such as a doubleword of zeroes)	DWORD	ULONG

MOU_DISPLAYMODECHANGE (5Dh) - Returns

None.

MOU_DISPLAYMODECHANGE (5Dh) - Remarks

This call does not provide return parameters or support input parameters. If the pointer was hidden on the mode switch start (Function 51h), then it is redrawn.

This function is the complement to MOU_UPDATEDISPLAYMODE.

MOU_DISPLAYMODECHANGE (5Dh) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_DISPLAYMODECHANGE (5Dh)
Description:
Notification of Mode Switch Completion

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_GETBUTTONCOUNT (60h) - Query Number of Mouse Buttor

MOU_GETBUTTONCOUNT (60h) - Description

This function returns the number of buttons supported by the physical mouse device driver.

MOU_GETBUTTONCOUNT (60h) - Parameter Packet Format

None.

MOU_GETBUTTONCOUNT (60h) - Number Supported

Number Supported
The number of buttons supported by the physical mouse device driver. Return values are in the range of:

1	One-button support
2	Two-button support
3	Three-button support

MOU_GETBUTTONCOUNT (60h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns to the caller the number of buttons supported by the physical device driver. The format of the Data Packet is as follows:

Field	Length	C Datatype
Number Supported	WORD	USHORT

MOU_GETBUTTONCOUNT (60h) - Returns

This function returns to the caller the number of buttons supported by the physical mouse device driver.

MOU_GETBUTTONCOUNT (60h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETBUTTONCOUNT (60h)
Description:
Query Number of Mouse Buttons Supported

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

MOU_GETMICKEYCOUNT (61h) - Query Mouse Device Motion Se

MOU_GETMICKEYCOUNT (61h) - Description

This function returns the mouse device's motion sensitivity.

MOU_GETMICKEYCOUNT (61h) - Parameter Packet Format

None.

MOU_GETMICKEYCOUNT (61h) - Mickeys/Centimeter

Mickeys/Centimeter
The number of mickeys/centimeter supported by the mouse device. Return values are in the range of:

```
0 < number of mickeys/centimeter <= (32K - 1)
```

MOU_GETMICKEYCOUNT (61h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns the mouse device's motion sensitivity to the caller. The format of the Data Packet is as follows:

Field	Length	C Datatype
Mickeys/Centimeter	WORD	USHORT

MOU_GETMICKEYCOUNT (61h) - Returns

This function returns to the caller the motion sensitivity of the mouse device as represented by the number of mickeys/centimeter.

MOU_GETMICKEYCOUNT (61h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETMICKEYCOUNT (61h)
Description:
Query Mouse Device Motion Sensitivity

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

MOU_GETMOUSTATUS (62h) - Query Current Physical Mouse Dri

MOU_GETMOUSTATUS (62h) - Description

This function returns the current physical mouse device driver Status flags.

MOU_GETMOUSTATUS (62h) - Parameter Packet Format

None.

MOU_GETMOUSTATUS (62h) - Status Flags

Status Flags

Has the following bit level definitions:

High Byte	Bit settings are as follows:	
	Bits 7-2	Reserved. Set to 0.
	Bit 1	Set, if mouse data returned in mickeys
	Bit 0	Set, if the interrupt level pointer draw routine is not called.
Low Byte	Bit settings are as follows:	
	Bits 7-4	Reserved. Set to 0.
	Bit 3	Set, if pointer draw routine disabled by unsupported mode
	Bit 2	Set, if flush in progress
	Bit 1	Set, if block read in progress
	Bit 0	Set, if event queue busy with I/O.

A set bit is a value of 1.

MOU_GETMOUSTATUS (62h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns the current device Status flags to the caller. The format of the Data Packet is as follows:

Field	Length	C Datatype
Status Flags	WORD	USHORT

MOU_GETMOUSTATUS (62h) - Returns

This function returns to the caller the current physical mouse device driver Status flags.

MOU_GETMOUSTATUS (62h) - Remarks

This function is the complement to MOU_SETMOUSTATUS.

MOU_GETMOUSTATUS (62h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETMOUSTATUS (62h)
Description:
Query Current Physical Mouse Driver Status Flags

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_READEVENTQUE (63h) - Read Mouse Event Queue

MOU_READEVENTQUE (63h) - Description

This function reads the mouse event queue.

MOU_READEVENTQUE (63h) - Read Type

Read Type

Used only to determine the type of action taken, if no event queue data is available. The values are:

0	Block the process (Wait) until event data is available
1	Return a NULL record (No-Wait) for the request.

MOU_READEVENTQUE (63h) - Parameter Packet Format

Field	Length	C Datatype
Read Type	WORD	USHORT

MOU_READEVENTQUE (63h) - Event Mask

Event Mask
See MOU_GETEVENTMASK.

MOU_READEVENTQUE (63h) - Time

Time
Event time stamp in milliseconds.

MOU_READEVENTQUE (63h) - Row Position

Row Position
Row-coordinate pointer.

MOU_READEVENTQUE (63h) - Column Position

Column Position
Column-coordinate pointer.

MOU_READEVENTQUE (63h) - Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller a 10-byte mouse event queue record. A mouse event queue record has the following format:

Field	Length	C Datatype
Event Mask	WORD	USHORT
Time	DWORD	ULONG
Row Position	WORD	USHORT
Column Position	WORD	USHORT

MOU_READEVENTQUE (63h) - Returns

This function returns to the caller a mouse event from the mouse event (FIFO) queue.

MOU_READEVENTQUE (63h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_READEVENTQUE (63h)
Description:
Read Mouse Event Queue

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

MOU_GETQUESTATUS (64h) - Query Event Queue Status

MOU_GETQUESTATUS (64h) - Description

This function returns the current event queue status.

MOU_GETQUESTATUS (64h) - Parameter Packet Format

None.

MOU_GETQUESTATUS (64h) - Element Count

Element Count
The current number of event queue elements. The return value is a 1-WORD value in the range of:

```
0 <= value <= MaxNumQueueElements
```

MOU_GETQUESTATUS (64h) - Queue Size

Queue Size
A 1-WORD return value for the MaxNumQueueElements.

MOU_GETQUESTATUS (64h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns to the caller the following data:

Field	Length	C Datatype
Element Count	WORD	USHORT
Queue Size	WORD	USHORT

MOU_GETQUESTATUS (64h) - Returns

This function returns to the caller the current number of queued elements in the mouse event queue, and the maximum number of elements allowed in a mouse event queue.

MOU_GETQUESTATUS (64h) - Remarks

MaxNumQueueElements is derived from the CONFIG.SYS QSIZE=keyword input parameter.

MOU_GETQUESTATUS (64h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETQUESTATUS (64h)
Description:
Query Event Queue Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_GETEVENTMASK (65h) - Query Current Mouse Event Mask

MOU_GETEVENTMASK (65h) - Description

Returns the current mouse event mask.

MOU_GETEVENTMASK (65h) - Parameter Packet Format

None.

MOU_GETEVENTMASK (65h) - Event Mask

Event Mask
Has the following bit level definitions:

Bits 7-15	Reserved. Must be 0.
Bit 6	Set, if Button 3 is down.
Bit 5	Set, if motion, with Button 3 down.
Bit 4	Set, if Button 2 is down.
Bit 3	Set, if motion, with Button 2 down.
Bit 2	Set, if Button 1 is down.
Bit 1	Set, if motion, with Button 1 down.
Bit 0	Set, if all mouse motion, no buttons down.

A set bit has a value of 1.

MOU_GETEVENTMASK (65h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns to the caller a 1-WORD current mouse Event Mask. The format of the Data Packet is as follows:

Field	Length	C Datatype
Event Mask	WORD	USHORT

MOU_GETEVENTMASK (65h) - Returns

This function returns to the caller the current mouse event mask.

MOU_GETEVENTMASK (65h) - Remarks

The return values can be any valid combination of enabled/disabled event flags. This is the complement function for MOU_SETEVENTMASK.

MOU_GETEVENTMASK (65h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETEVENTMASK (65h)
Description:
Query Current Mouse Event Mask

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_GETSCALEFACTORS (66h) - Query Current Mouse Scaling

MOU_GETSCALEFACTORS (66h) - Description

Returns the current mouse scaling factors.

MOU_GETSCALEFACTORS (66h) - Parameter Packet Format

None.

MOU_GETSCALEFACTORS (66h) - Row Data

Row Data
The row-coordinate scaling factor.

MOU_GETSCALEFACTORS (66h) - Column Data

Column Data
The column-coordinate scaling factor. The scaling factor values are positive integers in the range of:

```
0 < value <= (32K-1)
```

MOU_GETSCALEFACTORS (66h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns to the caller the current mouse scaling factors. The format of the Data Packet is as follows:

Field	Length	C Datatype
Row Data	WORD	USHORT
Column Data	WORD	USHORT

MOU_GETSCALEFACTORS (66h) - Returns

This function returns to the caller the current mouse scaling factors.

MOU_GETSCALEFACTORS (66h) - Remarks

Scaling factors are ratio values that determine how much relative movement is necessary before the physical mouse device driver reports a mouse event. The ratios specify the number of mickeys per 8 pels. The default ratio values are:
Vertical/Row ratio - 16 mickeys per 8 pels
Horizontal/Column ratio - 8 mickeys per 8 pels

This is the complement function for MOU_SETSCALEFACTORS.

MOU_GETSCALEFACTORS (66h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETSCALEFACTORS (66h)
Description:
Query Current Mouse Scaling Factors

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_GETPTRPOS (67h) - Query Current Pointer Screen Position

MOU_GETPTRPOS (67h) - Description

Returns the current pointer screen position.

MOU_GETPTRPOS (67h) - Parameter Packet Format

None.

MOU_GETPTRPOS (67h) - Row Position

Row Position
The row-coordinate pointer screen position.

MOU_GETPTRPOS (67h) - Column Position

Column Position
The column-coordinate pointer screen position.

MOU_GETPTRPOS (67h) - Data Packet Format

The Data Packet is a location in application storage where the physical mouse device driver returns to the caller the current pointer screen position. The format of the Data Packet is as follows:

Field	Length	C Datatype
Row Position	WORD	USHORT
Column Position	WORD	USHORT

MOU_GETPTRPOS (67h) - Returns

This function returns to the caller the current pointer screen position.

MOU_GETPTRPOS (67h) - Remarks

The coordinate values are display-mode dependent. Pel values are returned if the display is in graphics mode. Character position values are returned if the display is in text mode.

MOU_GETPTRPOS (67h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETPTRPOS (67h)
Description:
Query Current Pointer Screen Position

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_GETPTRSHAPE (68h) - Query Current Pointer Shape

MOU_GETPTRSHAPE (68h) - Description

Returns the current pointer shape.

MOU_GETPTRSHAPE (68h) - Buffer Length

Buffer Length
The length of the (user) Pointer Shape buffer in bytes. When calling this function, the caller must specify the length of the Pointer Shape buffer in this field.

MOU_GETPTRSHAPE (68h) - Columns

Columns
The width, in columns, of the pointer image.

MOU_GETPTRSHAPE (68h) - Rows

Rows
The height, in rows, of the pointer image.

MOU_GETPTRSHAPE (68h) - Column Hot Spot

Column Hot Spot
The column offset within the pointer image to the hot spot.

MOU_GETPTRSHAPE (68h) - Row Hot Spot

Row Hot Spot
The row offset within the pointer image to the hot spot.

MOU_GETPTRSHAPE (68h) - Parameter Packet Format

The Parameter Packet is a location in application storage where the physical mouse device driver returns the current pointer shape, a user buffer described by the data in the Data Packet.

Field	Length	C Datatype
Buffer Length	WORD	USHORT
Columns	WORD	USHORT
Rows	WORD	USHORT
Column Hot Spot	WORD	USHORT
Row Hot Spot	WORD	USHORT

MOU_GETPTRSHAPE (68h) - Data Packet Format

The Data Packet is a location in application storage containing the Pointer Definition record, which describes a user buffer where the physical mouse device driver returns the current pointer shape. This data has the following format:

MOU_GETPTRSHAPE (68h) - Returns

If the *Buffer* Length value specified by the caller is smaller than the required storage for the pointer shape, the physical mouse device driver returns an error. The physical mouse device driver returns the minimum storage requirements for the pointer shape in the *Buffer* Length field.

If the *Buffer* Length value specified by the caller is greater than, or equal to, the amount of storage required for the pointer shape, the physical mouse device driver returns the pointer shape in the user's Pointer Shape buffer (that is, in the Data Packet) and the data describing the pointer shape in the Pointer Definition record (Parameter Packet). The actual returned length is returned in the *Buffer* Length field.

MOU_GETPTRSHAPE (68h) - Remarks

The Pointer Shape buffer is described by the Pointer Definition record, and for normal conditions, consists of the Screen AND and Pointer XOR masks.

MOU_GETPTRSHAPE (68h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_GETPTRSHAPE (68h)
Description:
Query Current Pointer Shape

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

MOU_QUERYTHRESHOLDVALUES (69h) - Query Mouse Thresho

MOU_QUERYTHRESHOLDVALUES (69h) - Description

Returns the mouse acceleration parameters.

MOU_QUERYTHRESHOLDVALUES (69h) - Parameter Packet For

None.

MOU_QUERYTHRESHOLDVALUES (69h) - Length

Length
This should be initialized with the size of the data packet in bytes. If the length field is less than 10, then only the requested number of bytes are returned.

MOU_QUERYTHRESHOLDVALUES (69h) - 1st Movement Level

1st Movement Level
This value determines when the first level multiplier is applied.

MOU_QUERYTHRESHOLDVALUES (69h) - 1st Level Multiplier

1st Level Multiplier
This value is multiplied with the actual number of units moved to produce acceleration.

MOU_QUERYTHRESHOLDVALUES (69h) - 2nd Movement Level

2nd Movement Level
This values determines when the second level multiplier is applied.

MOU_QUERYTHRESHOLDVALUES (69h) - 2nd Level Multiplier

2nd Level Multiplier
This value is multiplied with the actual number of units moved to produce acceleration.

MOU_QUERYTHRESHOLDVALUES (69h) - Data Packet Format

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Length	WORD	USHORT
1st Movement Level	WORD	USHORT
1st Level Multiplier	WORD	USHORT
2nd Movement Level	WORD	USHORT
2nd Level Multiplier	WORD	USHORT

MOU_QUERYTHRESHOLDVALUES (69h) - Returns

Fills the data packet with the current threshold values.

MOU_QUERYTHRESHOLDVALUES (69h) - Remarks

For each mouse event, the count of physical units moved is compared against the 1st and 2nd level thresholds. If the count is greater than the 2nd level threshold, the 2nd level multiplier will be used; if the count is greater than the 1st level but less than the 2nd level, the 1st level multiplier will be used. If the count is less than the 1st level threshold, no multiplier will be used. These calculations are done separately for the X- and Y-axis.

MOU_QUERYTHRESHOLDVALUES (69h) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_QUERYTHRESHOLDVALUES (69h)
Description:
Query Mouse Threshold Values

Select an item:
Description
Parameter Packet Format

Data Packet Format
Returns
Remarks

MOU_VER (6Ah) - Query Physical Mouse Device Driver Level/Versi

MOU_VER (6Ah) - Description

Returns the physical mouse device driver level/version number.

MOU_VER (6Ah) - Parameter Packet Format

None.

MOU_VER (6Ah) - Version Number

Version Number
Has the following settings:

1	OS/2 1.3 level support for device-dependent-device drivers
2	OS/2 2.0 or 2.1 level support for device-dependent-device drivers

MOU_VER (6Ah) - Data Packet Format

The Data Packet is a WORD in application storage where the physical mouse device driver returns its version number.

Field	Length	C Datatype
Version Number	WORD	USHORT

MOU_VER (6Ah) - Returns

None.

MOU_VER (6Ah) -

Category:
IOCTL_POINTINGDEVICE (07h)
Function:
MOU_VER (6Ah)
Description:
Query Physical Mouse Device Driver Level/Version Number

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

MOU_QUERYPOINTERID (6Bh) - Query Pointing Device ID

MOU_QUERYPOINTERID (6Bh) - Description

Returns the Pointing Device ID.

MOU_QUERYPOINTERID (6Bh) - Parameter Packet Format

None.

MOU_QUERYPOINTERID (6Bh) - Device ID

Device ID
The ID number for the primary pointing device.

MOU_QUERYPOINTERID (6Bh) - COM Port Number

COM Port Number
The number of the communications port connected to the serial pointing device. Valid COM Port Numbers are:

1	Communications Port 1
2	Communications Port 2
3	Communications Port 3

MOU_QUERYPOINTERID (6Bh) - Secondary Device ID

Secondary Device ID
The Pointing Device ID for the secondary device.

MOU_QUERYPOINTERID (6Bh) - Data Packet Format

The Data Packet is a location in application storage with the following format:

Field	Length	C Datatype
Device ID	WORD	USHORT
COM Port Number	WORD	USHORT
Secondary Device ID	WORD	USHORT

MOU_QUERYPOINTERID (6Bh) - Returns

This function fills the caller's Data Packet fields.

MOU_QUERYPOINTERID (6Bh) - Remarks

Device IDs are specific to the type of pointing device installed. The supported Pointing Device IDs are listed below:

0	Unknown
1	Bus mouse
2	Serial mouse
3	Inport mouse
4	PS/2-style pointing device
5	IBM 8516 Touch Display
6-655351	Reserved

Note: If a serial pointing device is not installed, the COM Port Number field is meaningless.

MOU_QUERYPOINTERID (6Bh) -

Category:
IOCTL_POINTINGDEVICE (07h)

Function:
MOU_QUERYPOINTERID (6Bh)
Description:
Query Pointing Device ID

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 08h Logical Disk Control IOCtl Commands

The following is a summary of Category 08h IOCtl Commands:

Function	Description
00h	Lock Drive
01h	Unlock Drive
02h	Redetermine Media
03h	Set Logical Map
04h	Begin Format
20h	Block Removable
21h	Query Logical Map
22h	Reserved
40h	Removable Media Control
43h	Set Device Parameters
44h	Write Logical Track
45h	Format and Verify Track
5Dh	Diskette Control
5Eh	Reserved
5Fh	Reserved
60h	Query Media Sense
63h	Query Device Parameters
64h	Read Logical Track
65h	Verify Logical Track
66h	Status

DSK_LOCKDRIVE (00h) - Lock Drive

DSK_LOCKDRIVE (00h) - Description

This function locks a drive and is used to exclude I/O by another process on the volume in the drive. The Lock Drive function succeeds only if there is one file handle open on the volume in the drive (that is, the file handle on which this function is issued). This is necessary because the desired result is to exclude all other I/O to this volume until "Function 01h - Unlock Drive" is issued.

DSK_LOCKDRIVE (00h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_LOCKDRIVE (00h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_LOCKDRIVE (00h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	BYTE	UCHAR

DSK_LOCKDRIVE (00h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_LOCKDRIVE (00h) - Remarks

This function should be called after obtaining a handle from DosOpen but before actually accessing the drive. Notice that it is necessary to call DSK_UNLOCKDRIVE before closing the drive.

DSK_LOCKDRIVE (00h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_LOCKDRIVE (00h)
Description:
Lock Drive

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_UNLOCKDRIVE (01h) - Unlock Drive

DSK_UNLOCKDRIVE (01h) - Description

This function locks the drive.

DSK_UNLOCKDRIVE (01h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_UNLOCKDRIVE (01h) - Parameter Packet Format

Field	Length	C Datatype

Command Information	BYTE	UCHAR

DSK_UNLOCKDRIVE (01h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	BYTE	UCHAR

DSK_UNLOCKDRIVE (01h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_UNLOCKDRIVE (01h) - Remarks

This function releases a lock on a volume in a drive, thereby allowing I/O from other processes to that volume again. The locked volume represented by the file handle on which this function is issued, must be in the drive.

Notice that this function must be called after accessing a drive that was locked with DSK_LOCKDRIVE before closing the drive with DosClose.

DSK_UNLOCKDRIVE (01h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_UNLOCKDRIVE (01h)
Description:
Unlock Drive

Select an item:
Description
Parameter Packet Format
Data Packet Format

Returns
Remarks

DSK_REDETERMINEMEDIA (02h) - Redetermine Media

DSK_REDETERMINEMEDIA (02h) - Description

This function redetermines media (ends format).

DSK_REDETERMINEMEDIA (02h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_REDETERMINEMEDIA (02h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_REDETERMINEMEDIA (02h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	BYTE	UCHAR

DSK_REDETERMINEMEDIA (02h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_REDETERMINEMEDIA (02h) - Remarks

This function rebuilds the device parameters (including the Volume Parameter Block (VPB) used by the operating system to identify the volume), simulates a close on the current device handle, and forces a mount on the volume.

Function 02h dismounts the volume from the current file system drive (FSD), attaches the new FSD, and reattaches the current handle to the volume's new FSD. It is typically called after a new boot sector has been written to a volume (after a format has been done). The caller must have the disk or diskette locked when calling this function; otherwise, the function fails with the error ERROR_LOCK_VIOLATION.

The caller can have only one file open to refer to the disk or diskette. If other processes have the volume open or the calling process has opened the volume multiple times, the call fails and ERROR_DRIVE_BUSY is returned.

DSK_REDETERMINEMEDIA (02h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_REDETERMINEMEDIA (02h)
Description:
Redetermine Media

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_SETLOGICALMAP (03h) - Set Logical Map

DSK_SETLOGICALMAP (03h) - Description

This function sets the next logical drive letter than is used to reference the drive.

DSK_SETLOGICALMAP (03h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_SETLOGICALMAP (03h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_SETLOGICALMAP (03h) - Logical Drive Number

Logical Drive Number
On entry, a logical drive number (1=A, 2=B, and so forth) is specified. On return, this byte specifies the logical drive currently mapped to the drive that the specified file handle is opened on. A zero is returned if there is only one logical drive mapped onto this physical drive.

DSK_SETLOGICALMAP (03h) - Data Packet Format

Field	Length	C Datatype
Logical Drive Number	BYTE	UCHAR

DSK_SETLOGICALMAP (03h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_SETLOGICALMAP (03h) - Remarks

When copying diskettes on a drive whose physical drive number has more than one logical drive letter assigned to it, the operating system issues *diskette swap* prompts to tell the user which logical drive letter is currently referencing the physical drive number. Function 03h is typically used to avoid this prompt by setting a drive number (corresponding to the drive letter) that is referenced in the next I/O request. The last logical drive letter assigned to the physical drive is determined by calling DSK_GETLOGICALMAP.

DSK_SETLOGICALMAP (03h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_SETLOGICALMAP (03h)
Description:
Set Logical Map

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_BEGINFORMAT (04h) - Begin Format

DSK_BEGINFORMAT (04h) - Description

This function begins the format.

DSK_BEGINFORMAT (04h) - FSD Name

FSD Name
The file system driver name (that is, the name the FSD exports). A zero length string is used to indicate the FAT file system.

DSK_BEGINFORMAT (04h) - Parameter Packet Format

Field	Length	C Datatype
FSD Name	ASCIIZ string	PSZ

DSK_BEGINFORMAT (04h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_BEGINFORMAT (04h) - Data Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_BEGINFORMAT (04h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_BEGINFORMAT (04h) - Remarks

This function attaches (mounts) a specified File System Driver (FSD) to a logical disk volume. Function 04h is typically used to force mount the FSD that is formatting a volume. It is called before starting a format operation on a volume. This function also unmounts a current FSD (if any), and forces a mount to the specified FSD.

A flag is set in the OS/2 kernel to indicate that a Begin Format (unmount/mount) was done. DSK_REDETERMINEMEDIA clears this flag. Therefore, Function 02h must be performed before the disk is closed with DosClosed. The volume is then ready for normal I/O service from the newly mounted FSD.

DSK_BEGINFORMAT (04h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_BEGINFORMAT (04h)
Description:
Begin Format

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_BLOCKREMOVABLE (20h) - Block Removable

DSK_BLOCKREMOVABLE (20h) - Description

This function is used to determine if the media is removable or fixed.

DSK_BLOCKREMOVABLE (20h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_BLOCKREMOVABLE (20h) - Drive Unit

Drive Unit
This field is used only when this IOCtl is issued without using a previously allocated file handle. A file handle of -1 must be used. Note that that media in the drive is not required. Drive Unit values are 0 = A, 1 = B, 2 = C, and so forth.

DSK_BLOCKREMOVABLE (20h) - Parameter Packet Format

Field	Length	C Datatype

Command Information	BYTE	UCHAR
Drive Unit	BYTE	UCHAR

DSK_BLOCKREMOVABLE (20h) - Data

Data

On return, the data byte is set accordingly:

0	Removable media
1	Nonremovable media

DSK_BLOCKREMOVABLE (20h) - Data Packet Format

Field	Length	C Datatype
Data	BYTE	UCHAR

DSK_BLOCKREMOVABLE (20h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_BLOCKREMOVABLE (20h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_BLOCKREMOVABLE (20h)
Description:
Block Removable

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

DSK_GETLOGICALMAP (21h) - Query Logical Map

DSK_GETLOGICALMAP (21h) - Description

This function returns the logical drive letter that was last used to reference (open) the drive.

DSK_GETLOGICALMAP (21h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_GETLOGICALMAP (21h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_GETLOGICALMAP (21h) - Logical Drive Number

Logical Drive Number
On entry, a logical drive number (1=A, 2=B, and so forth) is specified. On return, if the device has more than one logical drive letter assigned to it, a drive number corresponding to the last drive letter that was used to reference the device is returned. For example, if the entry drive number was 1 (where 1=A) and the returned drive number was 2 (where 2=B), this drive was last referenced as the **B:** drive. If only one drive letter (logical drive) is assigned to the device, a zero is returned.

Note: This function works as long as the file handle is valid. The file handle can be set to anything other than zero.

DSK_GETLOGICALMAP (21h) - Data Packet Format

Field	Length	C Datatype
Logical Drive Number	BYTE	UCHAR

DSK_GETLOGICALMAP (21h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_GETLOGICALMAP (21h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_GETLOGICALMAP (21h)
Description:
Query Logical Map

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

DSK_UNLOCKEJECTMEDIA (40h) - Removable Media Control

DSK_UNLOCKEJECTMEDIA (40h) - Description

This function physically locks/unlocks the media in the drive. It also ejects the media for drives that support these functions.

DSK_UNLOCKEJECTMEDIA (40h) - Length Command Information

Length Command Information

Bits 0-1	Description
00	Unlock.
01	Lock - drive can be locked with or without media in it. If the drive is locked without media in it, the media will be ejected when it is inserted until an UnlockIOCtl is issued.
10	Eject Media - media will be ejected if the drive is not locked.
11	Loads Media.

DSK_UNLOCKEJECTMEDIA (40h) - Drive Unit

Drive Unit

This field is used only when the IOCtl is issued without using a previously allocated file handle. A file handle of -1 must be used. Notice that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

DSK_UNLOCKEJECTMEDIA (40h) - Parameter Packet Format

Field	Length	C Datatype
Length Command Information	BYTE	UCHAR
Drive Unit	BYTE	UCHAR

DSK_UNLOCKEJECTMEDIA (40h) - Data Packet Format

None.

DSK_UNLOCKEJECTMEDIA (40h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE

31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_UNLOCKEJECTMEDIA (40h) - Remarks

Not all floppy drives support these functions.

DSK_UNLOCKEJECTMEDIA (40h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_UNLOCKEJECTMEDIA (40h)
Description:
Removable Media Control

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_SETDEVICEPARAMS (43h) - Set Device Parameters

DSK_SETDEVICEPARAMS (43h) - Description

This function sets the parameters for a specified device.

DSK_SETDEVICEPARAMS (43h) - Command Information

Command Information
The two low bits of the command byte are used to indicate one of three possible actions:

Bits	Description
00	Revert to building the BIOS Parameter Block (BPB) off the medium for all subsequent Build BPB functions. This is used after a format operation to reset the device parameters to their original state.

01	Change the default BPB for the physical device. This changes the physical parameters *for* the drive as opposed to parameters for the media *in* the drive.
10	Change the BPB for the medium to the specified BPB, and return the new BPB for the medium for all subsequent Build BPB calls. This is used to prepare the device for a format media operation according to the device parameters specified.

All other bits are reserved, and must be set to 0.

DSK_SETDEVICEPARAMS (43h) - Drive Unit

Drive Unit

The field is used only when this IOCtl is issued without using a previously allocated file handle and when Command Information is set to 01. A file handle of -1 must be used. Note that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

DSK_SETDEVICEPARAMS (43h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR
Drive Unit	BYTE	UCHAR

DSK_SETDEVICEPARAMS (43h) - Extended BPB for Devices

Extended BPB for Devices

The term *Extended BPB* refers to the BIOS Parameter Block. It is a table that describes the structure of the media and the physical layout of the drive (heads, tracks, sectors).

The Extended BPB has the following format:

Field	Length
Bytes per Sector	WORD
Sectors per Cluster	BYTE
Reserved Sectors	WORD
Number of FATs	BYTE
Root Directory Entries	WORD
Total Sectors	WORD
Media Descriptor	BYTE
Sectors per FAT	WORD

Sectors per Track	WORD
Number of Heads	WORD
Hidden Sectors	DWORD
Large Total Sectors	DWORD
Reserved	6 BYTES

DSK_SETDEVICEPARAMS (43h) - Number of Cylinders

Number of Cylinders
Indicates the number of cylinders defined for the physical device.

DSK_SETDEVICEPARAMS (43h) - Device Type

Device Type
Describes the physical layout of the device specified, and has one of the following values:

0	48 TPI low-density diskette drive
1	96 TPI high-density diskette drive
2	Small (3.5-inch) 720KB drive
3	8-inch single-density diskette drive
4	8-inch double-density diskette drive
5	Fixed disk
6	Tape drive
7	Other (includes 1.44MB 3.5-inch diskette drive)
8	R/W optical disk
9	3.5-inch 4.0MB diskette drive (2.88MB formatted)

DSK_SETDEVICEPARAMS (43h) - Device Attributes

Device Attributes
A bit field that returns flag information about the specified drive:

Bit 0	Removable Media flag. This bit is set to 1 if the media cannot be removed. It is set to 0 if the media is removable.
Bit 1	Changeline flag. This bit is set to 1 if the device support determines that the media was removed since the last I/O operation. To query whether the media has changed, call the device driver Strategy Command "1h - MEDIA CHECK". (See "Physical Device Driver Strategy Commands" in the *Physical Device Driver Reference* for more information.) If this bit is set to 0, then the physical device driver can return the value 0, *Unsure if media has changed*, from the Media Check function.

DSK_SETDEVICEPARAMS (43h) - Data Packet Format

Field	Length	C Datatype
Extended BPB for Devices	31 BYTES	BYTE[31]
Number of Cylinders	WORD	USHORT
Device Type	BYTE	BYTE
Device Attributes	WORD	USHORT

Related C Data Structure

The BIOSPARAMETERBLOCK data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

DSK_SETDEVICEPARAMS (43h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_SETDEVICEPARAMS (43h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_SETDEVICEPARAMS (43h)
Description:
Set Device Parameters

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

DSK_WRITETRACK (44h) - Write Logical Track

DSK_WRITETRACK (44h) - Description

This function writes a logical track.

DSK_WRITETRACK (44h) - Command Information

Command Information
A byte with bit 0 defined as follows:

Bit 0	If clear (0), the track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

DSK_WRITETRACK (44h) - Head

Head
The physical head on the drive that performs the operation.

DSK_WRITETRACK (44h) - Cylinder

Cylinder
The cylinder for the write.

DSK_WRITETRACK (44h) - First Sector

First Sector
The logical sector number within the Track Layout Table that starts the I/O operation. Note that the sector numbers start with 0. For example, the third sector is number 2.

DSK_WRITETRACK (44h) - Number of Sectors

Number of Sectors
The number of sectors to write (up to the maximum specified in the track table; the IOCtl subfunctions do not step heads/tracks).

DSK_WRITETRACK (44h) - Track Layout Table

Track Layout Table

Has the following:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
.	.
.	.
.	.
Sector number for Sector *n*	WORD
Sector size for Sector *n*	WORD

The sector table that is specified provides information that is used during the WRITE track operation.

DSK_WRITETRACK (44h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR
Head	WORD	USHORT
Cylinder	WORD	USHORT
First Sector	WORD	USHORT
Number of Sectors	WORD	USHORT
Track Layout Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

DSK_WRITETRACK (44h) - Data Packet Format

The Data Packet is a buffer. It contains the data to be written.

Field	Length	C Datatype
Buffer	BYTES	UCHAR[]

DSK_WRITETRACK (44h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_WRITETRACK (44h) - Remarks

This function is used to perform a write of sectors on the media and to perform the operations on the device that is specified in this request. The track table passed in on the call is used to determine the sector number, which is passed to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive, this request breaks into *n* single sector operations and writes one sector at a time. This DosDevIOCtls is typically used when performing I/O outside of the normal file system data area on the media.

DSK_WRITETRACK (44h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_WRITETRACK (44h)
Description:
Write Logical Track

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_FORMATVERIFY (45h) - Format and Verify Track

DSK_FORMATVERIFY (45h) - Description

This function formats and verifies a track.

DSK_FORMATVERIFY (45h) - Command Information

Command Information

A byte with bit 0 defined as follows:

Bit 0	If clear (0), the track layout contains nonconsecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.
Bit 1	Indicates that this is a multi-track request. For multi-track requests, the track table should contain a single entry.

All other bits are reserved and must be set to 0.

DSK_FORMATVERIFY (45h) - Head

Head

The physical head on the drive that performs the operation.

DSK_FORMATVERIFY (45h) - Cylinder

Cylinder

The cylinder for the operation. Upon return from a multitrack format request, if a defective spot is encountered on the media, the returned head and cylinder contain the defective area.

DSK_FORMATVERIFY (45h) - Number of Tracks

Number of Tracks

The number of tracks to format/verify on a multitrack request. This is set to zero for single track requests. On return from a multitrack request, Number of Tracks is set to one of the following values:

Value	Description
8000H	Multitrack successful
4000H	Multitrack not supported
2000H	Multitrack failed on FORMAT operation
1000H	Multitrack failed on VERIFY operation

DSK_FORMATVERIFY (45h) - Number of Sectors

Number of Sectors

The number of sectors on the track being formatted. On return from a multitrack request, if a defective spot is found on the media before the multitrack format operation can complete, this field indicates the number of tracks remaining (to be formatted) on this multitrack format request.

DSK_FORMATVERIFY (45h) - Format Track Table

Format Track Table

Contains four-byte tuples. Each *tuple* is in the form (c, h, r, n) where *c*=cylinder number, *h*=head number, *r*=Sector ID, and *n*=save bytes per sector:

n	Bytes/Sector
0	128
1	256
2	512
3	1024

There is a 4-tuple for each sector in the track to be formatted. Both the head and cylinder numbers must be consistent within the tuple and the corresponding Parameter Packet field.

DSK_FORMATVERIFY (45h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR
Head	WORD	USHORT
Cylinder	WORD	USHORT
Number of Tracks	WORD	USHORT
Number of Sectors	WORD	USHORT
Format Track Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

DSK_FORMATVERIFY (45h) - Starting Sector

Starting Sector

On input, this is the starting sector on a multitrack request. This is set to zero for a single track request. On return from a multitrack format request, if a defective spot is found on the media, Starting Sector is the first logical sector number within the head and cylinder of the defective area.

DSK_FORMATVERIFY (45h) - Data Packet Format

Field	Length	C Datatype
Starting Sector	BYTE	BYTE

DSK_FORMATVERIFY (45h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_FORMATVERIFY (45h) - Remarks

The following describes a format track table for a track layout starting with sector 1 and containing only consecutive sectors (command information Bit 0 set to 1). This table also describes only a single track request (number of tracks set to 0) for a track with 12 sectors (number of sectors) formatted to 512 bytes-per-sector.

C	H	S	N
10	0	1	2
10	0	2	2
10	0	3	2
10	0	4	2
10	0	5	2
10	0	6	2
10	0	7	2
10	0	8	2

10	0	9	2
10	0	10	2
10	0	11	2
10	0	12	2

This routine formats and verifies the track specified according to the information passed in the Format Track Table. The track is passed to the controller, which performs the formatting. Note that some controllers do not support formatting tracks with varying sector sizes; therefore, applications must ensure that the sector sizes specified in the Track Layout Table are all the same.

0	0	1	2

DSK_FORMATVERIFY (45h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_FORMATVERIFY (45h)
Description:
Format and Verify Track

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_DISKETTECONTROL (5Dh) - Diskette Control

DSK_DISKETTECONTROL (5Dh) - Description

This function allows cooperative sharing of the diskette controller hardware by a separate device driver.

DSK_DISKETTECONTROL (5Dh) - Command Information

Command Information
Indicates whether the application is requesting or returning exclusive ownership of the diskette controller, or querying the diskette device driver.

Bits 0-1	Description

00	Resume - This IOCtl resumes processing of diskette requests by the diskette driver.
01	Suspend - When this IOCtl returns without error, the diskette driver will have relinquished control of the diskette hardware, including the IRQ level. Further requests to the diskette device from the operating system will be queued.
10	Query - This IOCtl indicates whether the diskette driver is idle (no pending requests in the device driver queue). A return code of 0 (NO_ERROR) indicates the device driver is idle.

DSK_DISKETTECONTROL (5Dh) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_DISKETTECONTROL (5Dh) - Reserved

Reserved
Reserved. Set to 0.

DSK_DISKETTECONTROL (5Dh) - Data Packet Format

Field	Length	C Datatype
Reserved	BYTE	UCHAR

DSK_DISKETTECONTROL (5Dh) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA

DSK_DISKETTECONTROL (5Dh) - Remarks

If pending requests are detected, then the handling is up to the tape application. It is recommended that the tape application do an orderly clean-up of its activities and then issue a RESUME IOCTL to allow diskette requests to proceed and subsequently issue another SUSPEND IOCTL to reclaim the diskette hardware.

DSK_DISKETTECONTROL (5Dh) -

Category:
IOCTL_DISK (08h)
Function:
DSK_DISKETTECONTROL (5Dh)
Description:
Diskette Control

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_QUERYMEDIASENSE (60h) - Query Media Sense

DSK_QUERYMEDIASENSE (60h) - Description

This function returns the media sense information.

DSK_QUERYMEDIASENSE (60h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_QUERYMEDIASENSE (60h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DSK_QUERYMEDIASENSE (60h) - Media Sense Information

Media Sense Information
On return, this field can be interpreted as follows:

0	Unable to determine media type
1	720KB diskette is present in 3.5-inch drive
2	1.44MB diskette is present in 3.5-inch drive
3	2.88MB diskette is present in 3.5-inch drive

DSK_QUERYMEDIASENSE (60h) - Data Packet Format

Field	Length	C Datatype
Media Sense Information	BYTE	UCHAR

DSK_QUERYMEDIASENSE (60h) - Returns

The error return codes for this function are as follows:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
22	ERROR_BAD_COMMAND
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_QUERYMEDIASENSE (60h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_QUERYMEDIASENSE (60h)

Description:
Query Media Sense

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

DSK_GETDEVICEPARAMS (63h) - Query Device Parameters

DSK_GETDEVICEPARAMS (63h) - Description

Returns the device parameters.

DSK_GETDEVICEPARAMS (63h) - Command Information

Command Information
A byte with bit 0 defined as follows:

0	Return the recommended BPB for the drive, which is the BPB for the physical device, unless it is a formatted fixed media. Then, it is the BPB that was on the media when the system was booted.
1	Return the BPB for the media currently in the drive. This always reads the BPB off the current media in the drive. An error is returned if the media is unformatted.

All other bits are reserved, and must be set to 0.

DSK_GETDEVICEPARAMS (63h) - Drive Unit

Drive Unit
Used only when this IOCtl is issued without using a previously allocated file handle and when Command Information is set to 0. A file handle of -1 must be used. Notice that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

DSK_GETDEVICEPARAMS (63h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR
Drive Unit	BYTE	UCHAR

DSK_GETDEVICEPARAMS (63h) - Extended BPB for Device

Extended BPB for Device

The physical device driver maintains two BPBs for each drive. One is the current BPB that corresponds to the media in the drive. The other is a recommended BPB based on the type of media that corresponds to the physical device. (For example, for a high-density drive, the BPB is for a 96 tracks-per-inch (TPI); for a low-density drive, the BPB is for a 48 TPI). The low bit of the Command Information field indicates which BPB the application needs to see.

DSK_GETDEVICEPARAMS (63h) - Number of Cylinders

Number of Cylinders

Indicates the number of cylinders defined for the physical device.

DSK_GETDEVICEPARAMS (63h) - Device Type

Device Type

Describes the physical layout of the device specified, and has one of the following values:

0	48 TPI low-density diskette drive
1	96 TPI high-density diskette drive
2	3.5-inch 720KB diskette drive
3	8-Inch single-density diskette drive
4	8-Inch double-density diskette drive
5	Fixed disk
6	Tape drive
7	Other (includes 1.44MB 3.5-inch diskette drive)
8	R/W optical disk
9	3.5-inch 4.0MB diskette drive (2.88MB formatted)

DSK_GETDEVICEPARAMS (63h) - Device Attributes

Device Attributes

A bit field that returns flag information about the specified drive:

Bit 0	Removable Media flag. This bit is set to 1 if the media cannot be removed. It is set to 0 if the media is removable.
Bit 1	Changeline flag. This bit is set to 1, if the device support determines that the media was removed since the last I/O operation. To query whether the media has changed, call the physical device driver Strategy Command "1h - MEDIA CHECK". (Refer to "Physical Device Driver Strategy Commands" in the *Physical Device Driver Reference* for more information.) If this bit is set to 0, then the physical device driver can return the value 0, *Unsure if media has changed*, from the Media Check function.

Bit 2	Greater than 16MB Support flag. If this bit is set to 1, the physical device driver supports physical addresses greater than 16MB.

DSK_GETDEVICEPARAMS (63h) - Data Packet Format

Field	Length	C Datatype
Extended BPB for Device	31 BYTES	UCHAR[31]
Number of Cylinders	WORD	USHORT
Device Type	BYTE	UCHAR
Device Attributes	WORD	USHORT

DSK_GETDEVICEPARAMS (63h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_GETDEVICEPARAMS (63h) - Remarks

This function gets the parameters for a specified device.

DSK_GETDEVICEPARAMS (63h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_GETDEVICEPARAMS (63h)
Description:
Query Device Parameters

Select an item:
Description
Parameter Packet Format

Data Packet Format
Returns
Remarks

DSK_READTRACK (64h) - Read Logical Track

DSK_READTRACK (64h) - Description

Reads a logical track.

DSK_READTRACK (64h) - Command Information

Command Information

A byte with bit 0 defined as follows:

Bit 0	If clear (0), the track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

DSK_READTRACK (64h) - Head

Head

The physical head on the drive that performs the operation.

DSK_READTRACK (64h) - Cylinder

Cylinder

The cylinder for the read.

DSK_READTRACK (64h) - First Sector

First Sector

The logical sector number within the Track Layout Table that starts the I/O operation. Note that the sector numbers start with 0. For example, the third sector is number 2.

DSK_READTRACK (64h) - Number of Sectors

Number of Sectors

The number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions do not step heads/tracks).

DSK_READTRACK (64h) - Track Layout Table

Track Layout Table

Has the following:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
Sector Size for Sector 3	WORD
.	.
.	.
.	.
Sector number for Sector *n*	WORD
Sector size for Sector *n*	WORD

The sector table that is specified provides information that is used during the READ track operation.

DSK_READTRACK (64h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	BYTE
Head	WORD	USHORT
Cylinder	WORD	USHORT
First Sector	WORD	USHORT
Number of Sectors	WORD	USHORT

Track Layout Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

DSK_READTRACK (64h) - Data Packet Format

The Data Packet is a buffer. The buffer must be large enough to hold requested data.

Field	Length	C Datatype
Buffer	BYTES	UCHAR[]

DSK_READTRACK (64h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_READTRACK (64h) - Remarks

This function is used to perform a read of sectors on the media and to perform the operations on the device that is specified in this request. The track table passed in on the call is used to determine the sector number, which is passed to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive, this request breaks into *n* single sector operations and reads one sector at a time. This DosDevIOCtls is typically used when performing I/O outside of the normal file system data area on the media.

DSK_READTRACK (64h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_READTRACK (64h)
Description:
Read Logical Track

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DSK_VERIFYTRACK (65h) - Verify Logical Track

DSK_VERIFYTRACK (65h) - Description

Verifies a logical track.

DSK_VERIFYTRACK (65h) - Command Information

Command Information

A byte with bit 0 defined as follows:

Bit 0	If clear (0), the track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

DSK_VERIFYTRACK (65h) - Head

Head

The physical head on the drive that performs the operation.

DSK_VERIFYTRACK (65h) - Cylinder

Cylinder

The cylinder for the verify.

DSK_VERIFYTRACK (65h) - First Sector

First Sector

The logical sector number within the Track Layout Table that starts the I/O operation. Note that the sector numbers start with 0. For example, the third sector is number 2.

DSK_VERIFYTRACK (65h) - Number of Sectors

Number of Sectors

The number of sectors to verify (up to the maximum specified in the track table; the IOCtl subfunctions do not step heads/tracks).

DSK_VERIFYTRACK (65h) - Track Layout Table

Track Layout Table

Has the following:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
Sector Size for Sector 3	WORD
.	.
.	.
.	.
Sector number for Sector *n*	WORD
Sector size for Sector *n*	WORD

The sector table that is specified provides information that is used during the VERIFY track operation.

DSK_VERIFYTRACK (65h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	BYTE
Head	WORD	USHORT
Cylinder	WORD	USHORT
First Sector	WORD	USHORT

Number of Sectors	WORD	USHORT
Track Layout Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

DSK_VERIFYTRACK (65h) - Data Packet Format

None.

DSK_VERIFYTRACK (65h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_VERIFYTRACK (65h) - Remarks

This function is used to perform a verify of sectors on the media and to perform the operations on the device that is specified in this request. The track table passed in on the call is used to determine the sector number, which is passed to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive, this request breaks into *n* single sector operations and verifies one sector at a time. This DosDevIOCtls is typically used when performing I/O outside of the normal file system data area on the media.

DSK_VERIFYTRACK (65h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_VERIFYTRACK (65h)
Description:
Verify Logical Track

Select an item:
Description
Parameter Packet Format
Data Packet Format

Returns
Remarks

DSK_GETLOCKSTATUS (66h) - Status

DSK_GETLOCKSTATUS (66h) - Description

The output of this function indicates if the drive is locked with or without media in it.

DSK_GETLOCKSTATUS (66h) - Command Information

Command Information
Reserved. Must be set to 0.

DSK_GETLOCKSTATUS (66h) - Drive Unit

Drive Unit
This field is used only when the IOCtl is issued without using a previously allocated file handle. A file handle of -1 must be used. Notice that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

DSK_GETLOCKSTATUS (66h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR
Drive Unit	WORD	USHORT

DSK_GETLOCKSTATUS (66h) - Status

Status

Bits 0-1	Value
00	Lock/Unlock/Eject/Status functions not supported.
01	Drive locked. Lock/Unlock/Eject functions supported.
10	Drive unlocked. Lock/Unlock/Eject functions supported.
11	Lock Status not supported. Lock/Unlock/Eject functions supported.
Bit 2	**Value**
0	No media in drive
1	Media in drive
Bit 3-31	Reserved

DSK_GETLOCKSTATUS (66h) - Data Packet Format

Field	Length	C Datatype
Status	WORD	USHORT

DSK_GETLOCKSTATUS (66h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

DSK_GETLOCKSTATUS (66h) - Remarks

None.

DSK_GETLOCKSTATUS (66h) -

Category:
IOCTL_DISK (08h)
Function:
DSK_GETLOCKSTATUS (66h)
Description:
Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 09h Physical Disk Control IOCtl Commands

Category 09h is used to access physical partitionable hard disks. The handle used for Category 09h command is returned by the DosPhysicalDisk (Function 2) API function. (See the *OS/2 Control Program Programming Reference* for more information). This handle is used to tell the system which physical disk is accessed by the IOCtl command.

The Physical Disk Control commands relate to the entire partitionable hard disk. Direct track and sector I/O start at the beginning of the physical drive. PDSK_GETPHYSDEVICEPARAMS, describes the entire physical device.

The following is a summary of Category 09h IOCtl Commands:

Function	Description
00h	Lock Physical Drive
01h	Unlock Physical Drive
44h	Write Physical Track
63h	Query Physical Device Parameters
64h	Read Physical Track
65h	Verify Physical Track

PDSK_LOCKPHYSDRIVE (00h) - Lock Physical Drive

PDSK_LOCKPHYSDRIVE (00h) - Description

This function locks the physical drive.

PDSK_LOCKPHYSDRIVE (00h) - Command Information

Command Information
Reserved. Must be set to 0.

PDSK_LOCKPHYSDRIVE (00h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PDSK_LOCKPHYSDRIVE (00h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	31 BYTES	UCHAR[31]

PDSK_LOCKPHYSDRIVE (00h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PDSK_LOCKPHYSDRIVE (00h) - Remarks

All the logical units on the physical drive are affected as well.

PDSK_LOCKPHYSDRIVE (00h) -

Category:
IOCTL_PHYSICALDISK (09h)
Function:
PDSK_LOCKPHYSDRIVE (00h)
Description:
Lock Physical Drive

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PDSK_UNLOCKPHYSDRIVE (01h) - Unlock Physical Drive

PDSK_UNLOCKPHYSDRIVE (01h) - Description

This function unlocks the physical drive.

PDSK_UNLOCKPHYSDRIVE (01h) - Command Information

Command Information
Reserved. Must be set to 0.

PDSK_UNLOCKPHYSDRIVE (01h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PDSK_UNLOCKPHYSDRIVE (01h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	31 BYTES	UCHAR[31]

PDSK_UNLOCKPHYSDRIVE (01h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PDSK_UNLOCKPHYSDRIVE (01h) - Remarks

All the logical units on the physical drive are affected as well.

PDSK_UNLOCKPHYSDRIVE (01h) -

Category:
IOCTL_PHYSICALDISK (09h)
Function:
PDSK_UNLOCKPHYSDRIVE (01h)
Description:
Unlock Physical Drive

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PDSK_WRITEPHYSTRACK (44h) - Write Physical Track

PDSK_WRITEPHYSTRACK (44h) - Description

This function performs a physical write track.

PDSK_WRITEPHYSTRACK (44h) - Command Information

Command Information

A bit field as follows:

Bit 0	If clear (0), the track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

PDSK_WRITEPHYSTRACK (44h) - Head

Head

The physical head on the drive that performs the operation.

PDSK_WRITEPHYSTRACK (44h) - Cylinder

Cylinder

The cylinder for the write.

PDSK_WRITEPHYSTRACK (44h) - First Sector

First Sector

The logical sector number within the Track Layout Table that starts the I/O operation. Note that the sector numbers start with 0. For example, the third sector is number 2.

PDSK_WRITEPHYSTRACK (44h) - Number of Sectors

Number of Sectors

The number of sectors to write (up to the maximum specified in the track table; the IOCtl subfunctions do not step heads/tracks).

PDSK_WRITEPHYSTRACK (44h) - Track Layout Table

Track Layout Table

The Track Layout Table is as follows:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
.	.
.	.
.	.
Sector number for Sector *n*	WORD
Sector size for Sector *n*	WORD

PDSK_WRITEPHYSTRACK (44h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	BYTE
Head	WORD	USHORT
Cylinder	WORD	USHORT
First Sector	WORD	USHORT
Number of Sectors	WORD	USHORT
Track Layout Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PDSK_WRITEPHYSTRACK (44h) - Data Packet Format

The Data Packet is a buffer. It contains the data to be written.

Field	Length	C Datatype
Buffer	BYTES	UCHAR[]

PDSK_WRITEPHYSTRACK (44h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PDSK_WRITEPHYSTRACK (44h) - Remarks

This function is used to perform the operations specified on the physical drive in this request. This is similar to the Category 08h commands, except that the I/O is done offset from the beginning of the physical drive instead of from the beginning of the extended volume associated with the unit number (Category 08h).

The Track Layout Table passed in the Parameter Packet is used to determine the sector number, which is passed on to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive, this request breaks into *n* single sector operations and writes one sector at a time. The sector table that is specified provides information that is used during the WRITE track operation.

PDSK_WRITEPHYSTRACK (44h) -

Category:
IOCTL_PHYSICALDISK (09h)
Function:
PDSK_WRITEPHYSTRACK (44h)
Description:
Write Physical Track

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PDSK_GETPHYSDEVICEPARAMS (63h) - Query Physical Device

PDSK_GETPHYSDEVICEPARAMS (63h) - Description

This function returns the physical device parameters.

PDSK_GETPHYSDEVICEPARAMS (63h) - Command Information

Command Information
Reserved. Must be set to 0.

PDSK_GETPHYSDEVICEPARAMS (63h) - Parameter Packet Form

Field	Length	C Datatype
Command Information	BYTE	UCHAR

PDSK_GETPHYSDEVICEPARAMS (63h) - Number of Cylinders

Number of Cylinders
The number of cylinders on the physical drive.

PDSK_GETPHYSDEVICEPARAMS (63h) - Number of Heads

Number of Heads
The number of heads on the physical drive.

PDSK_GETPHYSDEVICEPARAMS (63h) - Number of Sectors per

Number of Sectors per Track
The number of sectors per track on the physical drive.

PDSK_GETPHYSDEVICEPARAMS (63h) - Data Packet Format

Field	Length	C Datatype
Reserved	WORD	USHORT
Number of Cylinders	WORD	USHORT
Number of Heads	WORD	USHORT
Number of Sectors per Track	WORD	USHORT
Reserved	WORD	USHORT
Reserved	WORD	USHORT
Reserved	WORD	USHORT
Reserved	WORD	USHORT

Related C Data Structure

The DEVICEPARAMETERBLOCK data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PDSK_GETPHYSDEVICEPARAMS (63h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PDSK_GETPHYSDEVICEPARAMS (63h) - Remarks

The data values returned apply to the entire physical disk.

PDSK_GETPHYSDEVICEPARAMS (63h) -

Category:
IOCTL_PHYSICALDISK (09h)
Function:
PDSK_GETPHYSDEVICEPARAMS (63h)
Description:
Query Physical Device Parameters

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PDSK_READPHYSTRACK (64h) - Read Physical Track

PDSK_READPHYSTRACK (64h) - Description

Performs a physical read track.

PDSK_READPHYSTRACK (64h) - Command Information

Command Information

A bit field as follows:

Bit 0	If clear (0), the track layout contains nonconsecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

PDSK_READPHYSTRACK (64h) - Head

Head

The physical head on the drive that performs the operation.

PDSK_READPHYSTRACK (64h) - Cylinder

Cylinder

The cylinder for the read.

PDSK_READPHYSTRACK (64h) - First Sector

First Sector

The logical sector number within the Track Layout Table that starts the I/O operation. Note that the sector numbers start with 0. For example, the third sector is number 2.

PDSK_READPHYSTRACK (64h) - Number of Sectors

Number of Sectors

The number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions do not step heads/tracks).

PDSK_READPHYSTRACK (64h) - Track Layout Table

Track Layout Table

The Track Layout Table is as follows:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
Sector Size for Sector 3	WORD
.	.
.	.
.	.
Sector number for Sector *n*	WORD
Sector size for Sector *n*	WORD

PDSK_READPHYSTRACK (64h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	BYTE
Head	WORD	USHORT
Cylinder	WORD	USHORT
First Sector	WORD	USHORT
Number of Sectors	WORD	USHORT

Track Layout Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PDSK_READPHYSTRACK (64h) - Data Packet Format

The Data Packet is a buffer. The buffer must be large enough to hold requested data.

Field	Length	C Datatype
Buffer	BYTES	UCHAR[]

PDSK_READPHYSTRACK (64h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PDSK_READPHYSTRACK (64h) - Remarks

This function is used to perform the operations specified on the physical drive in this request. This is similar to the Category 08h commands, except that the I/O is done offset from the beginning of the physical drive instead of from the beginning of the extended volume associated with the unit number (Category 08h).

The Track Layout Table passed in the Parameter Packet is used to determine the sector number, which is passed on to the disk controller for the operation. In cases where the sectors are oddly numbered or are nonconsecutive, this request breaks into *n* single sector operations and reads one sector at a time. Note also that the device driver does not correctly read a non-512 byte sector if the READ operation would generate a DMA violation error. *Applications should be written so that this error does not occur.* The sector table that is specified provides information that is used during the READ track operation.

PDSK_READPHYSTRACK (64h) -

Category:

IOCTL_PHYSICALDISK (09h)
Function:
PDSK_READPHYSTRACK (64h)
Description:
Read Physical Track

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

PDSK_VERIFYPHYSTRACK (65h) - Verify Physical Track

PDSK_VERIFYPHYSTRACK (65h) - Description

Performs a physical verify track.

PDSK_VERIFYPHYSTRACK (65h) - Command Information

Command Information
A bit field as follows:

Bit 0	If clear (0), the track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), the track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

PDSK_VERIFYPHYSTRACK (65h) - Head

Head
The physical head on the drive that performs the operation.

PDSK_VERIFYPHYSTRACK (65h) - Cylinder

Cylinder
The cylinder for the verify.

PDSK_VERIFYPHYSTRACK (65h) - First Sector

First Sector

The logical sector number within the Track Layout Table that starts the I/O operation. Note that the sector numbers start with 0. For example, the third sector is number 2.

PDSK_VERIFYPHYSTRACK (65h) - Number of Sectors

Number of Sectors

The number of sectors to verify (up to the maximum specified in the track table; the IOCtl subfunctions do not step heads/tracks).

PDSK_VERIFYPHYSTRACK (65h) - Track Layout Table

Track Layout Table

The Track Layout Table is as follows:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
.	.
.	.
.	.
Sector number for Sector *n*	WORD
Sector size for Sector *n*	WORD

PDSK_VERIFYPHYSTRACK (65h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	BYTE

Head	WORD	USHORT
Cylinder	WORD	USHORT
First Sector	WORD	USHORT
Number of Sectors	WORD	USHORT
Track Layout Table	BYTES	BYTE[]

Related C Data Structure

The TRACKLAYOUT data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

PDSK_VERIFYPHYSTRACK (65h) - Data Packet Format

None.

PDSK_VERIFYPHYSTRACK (65h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

PDSK_VERIFYPHYSTRACK (65h) - Remarks

This function is used to perform the operations specified on the physical drive in this request. This is similar to the Category 08h commands, except that the I/O is done offset from the beginning of the physical drive instead of from the beginning of the extended volume associated with the unit number (Category 08h).

The Track Layout Table passed in the Parameter Packet is used to determine the sector number, which is passed on to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive, this request breaks into *n* single sector operations and verifies one sector at a time. The sector table that is specified provides information that is used during the VERIFY track operation.

PDSK_VERIFYPHYSTRACK (65h) -

Category:
IOCTL_PHYSICALDISK (09h)

Function:
PDSK_VERIFYPHYSTRACK (65h)
Description:
Verify Physical Track

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 0Ah Character Device Monitor IOCtl Command

The following is the Category 0Ah IOCtl Command:

Function	Description
40h	Register Monitor

MON_REGISTERMONITOR (40h) - Register Monitor

MON_REGISTERMONITOR (40h) - Description

This function registers a monitor.

MON_REGISTERMONITOR (40h) - Command Information

Command Information
Reserved. Must be set to 0.

MON_REGISTERMONITOR (40h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

MON_REGISTERMONITOR (40h) - Placement Flag

Placement Flag

The DosMonReg function call parameter that is used by an application to indicate:

- Where its monitor buffers are to be placed within a monitor chain relative to monitors already registered on the monitor chain
- What special processing requirements need to be supported by the monitor dispatcher.

Refer to the DosMonReg function description in the *OS/2 Control Program Programming Reference* for valid parameter values.

MON_REGISTERMONITOR (40h) - Index

Index

Used by an application to indicate on which monitor chain its monitor buffers are being registered. The accepted values for this parameter vary for each device driver. See information on the physical mouse device driver, the physical keyboard device driver and the physical parallel port device driver in the *OS/2 Input/Output Device Driver Reference* to determine the valid parameter value for each device driver.

Refer to the descriptions of the DosMonReg, DosMonRead, and DosMonWrite functions in the *OS/2 Control Program Programming Reference* for more information.

MON_REGISTERMONITOR (40h) - Address of Input Buffer

Address of Input Buffer

Specifies the address of a monitor input buffer allocated by an application, initialized by the monitor dispatcher, and used by the DosMonRead function.

MON_REGISTERMONITOR (40h) - Offset of Output Buffer

Offset of Output Buffer

Specifies the offset of a monitor output buffer allocated by an application in the same data segment as the input buffer, initialized by the monitor dispatcher, and used by the DosMonWrite function.

MON_REGISTERMONITOR (40h) - Data Packet Format

The Data Packet includes the following parameters passed from a DosMonReg function:

Field	Length	C Datatype

Placement Flag	WORD	USHORT
Index	WORD	USHORT
Address of Input Buffer	DWORD	PCHAR
Offset of Output Buffer	WORD	NPCHAR

Related C Data Structure

The MONITORPOSITION data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

MON_REGISTERMONITOR (40h) - Returns

An error is returned to the caller if monitor registration fails because of invalid parameters (for example, the specification of an invalid monitor chain (Index), the monitor buffers are too small, or there are not enough system resources).

When an error condition occurs, the following return codes are found in the request packet status field:

12H	NO_MONITOR_SUPPORT
03H	BAD_COMMAND
0CH	GENERAL_FAILURE

MON_REGISTERMONITOR (40h) - Remarks

Character device drivers that support device monitors receive this request when an application calls DosMonReg to register a monitor. The physical device driver places the values of these parameters in registers and calls the DevHlp_Register to complete monitor registration.

MON_REGISTERMONITOR (40h) -

Category:
IOCTL_MONITOR (0Ah)
Function:
MON_REGISTERMONITOR (40h)
Description:
Register Monitor

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 0Bh General Device Control IOCtl Commands

The following is a summary of Category 0Bh IOCtl Commands:

Function	Description

01h	Flush Input Buffer
02h	Flush Output Buffer
41h	System Notifications for Physical Device Drivers
60h	Query Monitor Support

DEV_FLUSHINPUT (01h) - Flush Input Buffer

DEV_FLUSHINPUT (01h) - Description

This function flushes the input buffer.

DEV_FLUSHINPUT (01h) - Command Information

Command Information
Reserved. Must be set to 0.

DEV_FLUSHINPUT (01h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DEV_FLUSHINPUT (01h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	WORD	USHORT

DEV_FLUSHINPUT (01h) - Returns

None.

DEV_FLUSHINPUT (01h) -

Category:
IOCTL_GENERAL (0Bh)
Function:
DEV_FLUSHINPUT (01h)
Description:
Flush Input Buffer

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

DEV_FLUSHOUTPUT (02h) - Flush Output Buffer

DEV_FLUSHOUTPUT (02h) - Description

This function flushes the output buffer.

DEV_FLUSHOUTPUT (02h) - Command Information

Command Information
Reserved. Must be set to 0.

DEV_FLUSHOUTPUT (02h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DEV_FLUSHOUTPUT (02h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	WORD	USHORT

DEV_FLUSHOUTPUT (02h) - Returns

None.

DEV_FLUSHOUTPUT (02h) -

Category:
IOCTL_GENERAL (0Bh)
Function:
DEV_FLUSHOUTPUT (02h)
Description:
Flush Output Buffer

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns

DEV_SYSTEMNOTIFYPDD (41h) - System Notifications for Physica

DEV_SYSTEMNOTIFYPDD (41h) - Description

This function notifies the physical device driver of any activities or modifications that occur during session switching and of any changes to the alternate-input-methods interfaces.

DEV_SYSTEMNOTIFYPDD (41h) - Length

Length

Contains the size of this structure, in bytes, including Length itself. Valid Length values are 8, 12, and 24. Length values of 8 and 12 are used for notifications pertaining to session information, such as session creation, termination, saving/restoring of sessions, and session switching. A length value of 24 is used for Alternate Input Methods (AIM) value verifications and activations. Notifications received with Length values other than those defined above should be returned with an ERROR_INVALID_PARAMETER error code.

DEV_SYSTEMNOTIFYPDD (41h) - Action

Action

Contains a value indicating the notification-calling condition. Action is a bit map that describes the type of notification made to the physical device driver. The bit-map definitions are as follows:

Bits 15-9	Reserved. Must equal 0.
Bit 8	Action = 256. AIM post-save activation
Bit 7	Action = 128. AIM pre-save verification
Bit 6	Action = 64. End of session switch
Bit 5	Action = 32. Start of session switch
Bit 4	Action = 16. Session creation
Bit 3	Action = 8. Session termination
Bit 2	Action = 4. Post-session restore
Bit 1	Action = 2. Post-session save
Bit 0	Action = 1. Pre-session save

Physical device drivers can register for any combination of these notification values by issuing the DOSSMRegisterDD API function. If a physical device driver receives a notification with an Action value it does not support, it returns an ERROR_INVALID_PARAMETER error code.

DEV_SYSTEMNOTIFYPDD (41h) - Call Data (number of bytes vari Action)

Call Data (number of bytes varies with the value of Action)

The combination of the Length field value and Action field values define the Call Data area size, format, and field definitions. The valid Call Data formats supported by this interface are defined as follows:

Length=8

Action=1. Pre-session save. Call Data contains the following fields to represent the outgoing session information:

Field	Length
Session Type	WORD
Session ID	WORD

Action=2. Post-session save. Call Data contains the following fields to represent the changing (incoming and outgoing Session IDs) session information:

Field	Length
Session ID In	WORD
Session ID Out	WORD

Action=4. Post-session restore. Call Data contains the following fields to represent the restored (incoming) session information:

Field	Length
Session ID	WORD
Session Type	WORD

Action=8. Session termination. Call Data contains the following fields to represent the terminating session information:

Field	Length
Session Type	WORD
Session ID	WORD

Action=16. Session creation. Call Data contains the following fields to represent the newly created session information:

Field	Length
Session ID	WORD
Session Type	WORD

Length=12

Action=32. Start of session switch. Call Data contains the following fields to represent session-switching (incoming and outgoing session) information:

Field	Length
Session In ID	WORD
Session In Type	WORD
Session Out ID	WORD
Session Out Type	WORD

Action=64. End of session switch. Call Data contains the following fields to represent session-switching information:

Field	Length
Session In ID	WORD
Session In Type	WORD
Session Out ID	WORD
Session Out Type	WORD

Length=24

Action=128. AIM pre-save verify. This type of notification is issued to device drivers to verify that the new AIM parameters are valid. The new values are not actually set until the AIM post-save activate notification.

Action=256. AIM post-save activate. The Call Data area format for Length=24 (all Action values) is defined as follows:

Field	Length
AIM_Errors	DWORD
AIM_Active	WORD
AIM_Timeout	WORD
AIM_FKAccept	DWORD
AIM_FKRate	DWORD
AIM_FKDelay	DWORD

AIM_Errors

Contains a return bit mask that identifies the specific AIM parameter value/field that caused an error condition. A set bit (value of 1) indicates an error. The AIM_Errors bit definitions are as follows:

Bits 31-5	Reserved. Must equal 0.
Bit 4	Set, if AIM_FKDelay value error
Bit 3	Set, if AIM_FKRate value error
Bit 2	Set, if AIM_FKAccept value error
Bit 1	Set, if AIM_Timeout value error
Bit 0	Set, if AIM_Active value error

Note: Device drivers performing AIM parameter validation must do so on the pre-save notification calls. Error return codes are not meaningful and are not returned on AIM post-save notification calls.

AIM_Active

Contains the Special Needs Status flag. Possible AIM_Active values are:

0	Indicates that AIM Support is currently inactive (FALSE).
1	Indicates that AIM Support is currently active (TRUE).

AIM_Timeout

Contains the number of seconds that the Special Needs methods should remain active once keyboard usage has ceased. A value of 0 indicates no timeout.

AIM_FKAccept

Contains the number of milliseconds a key

	must be pressed in order for it to be recognized as a *key press* event. A value of 0 indicates that no additional time is applied to the standard hardware response time.
AIM_FKRate	Indicates the number of milliseconds between recognized typematic key press events when a key is held down. A value of 0 indicates that no additional time is applied to the standard hardware rate.
AIM_FKDelay	Contains the number of milliseconds that a key press must be held down before generating typematic support with AIM_FKRate. A value of 0 indicates typematic keystroke support is disabled.

DEV_SYSTEMNOTIFYPDD (41h) - Parameter Packet Format

Field	Length	C Datatype
Length	WORD	USHORT
Action	WORD	USHORT
Call Data (number of bytes varies with the value of Action)	BYTES	UCHAR[]

DEV_SYSTEMNOTIFYPDD (41h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	WORD	USHORT

DEV_SYSTEMNOTIFYPDD (41h) - Returns

The error return codes for this function are as follows:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY

119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED
13H	INVALID_PARAMETER
0CH	GENERAL_FAILURE

DEV_SYSTEMNOTIFYPDD (41h) - Remarks

Session-switching and saving/restoring notifications are not given when switching between non-full screen sessions (that is, between Presentation Manager sessions and text-windowed sessions). Termination and creation notifications are given in all cases.

For requests with an Action parameter value in the range of 1-64, refer to the DosStartSession API in the *OS/2 Control Program Programming Reference* for Session ID/Type parameter values and descriptions. Any request received by a physical device driver that does not match the Length/Action parameter values, as defined above, is returned to the caller with an ERROR_INVALID_PARAMETER error code.

DEV_SYSTEMNOTIFYPDD (41h) -

Category:
IOCTL_GENERAL (0Bh)
Function:
DEV_SYSTEMNOTIFYPDD (41h)
Description:
System Notifications for Physical Device Drivers

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

DEV_QUERYMONSUPPORT (60h) - Query Monitor Support

DEV_QUERYMONSUPPORT (60h) - Description

This function flushes the output buffer.

DEV_QUERYMONSUPPORT (60h) - Command Information

Command Information
Reserved. Must be set to 0.

DEV_QUERYMONSUPPORT (60h) - Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR

DEV_QUERYMONSUPPORT (60h) - Data Packet Format

Field	Length	C Datatype
Reserved. Set to 0.	WORD	USHORT

DEV_QUERYMONSUPPORT (60h) - Returns

None.

DEV_QUERYMONSUPPORT (60h) - Remarks

This request is used to query a device driver for monitor support. The physical device driver returns the system error MONITORS_NOT_SUPPORTED if it does not support character monitors. If monitors are supported, then it returns NO_ERROR (00H).

DEV_QUERYMONSUPPORT (60h) -

Category:
IOCTL_GENERAL (0Bh)
Function:
DEV_QUERYMONSUPPORT (60h)
Description:
Query Monitor Support

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 0Ch Advanced Power Management

Function	Description
40h	Send Power Event
41h	Set Power Event Resource
42h - 44h	Reserved
45h	OEM APM Function
60h	Query Power Status
61h	Query Power Event
62h	Query Power Information
63h	Query Power State

POWER_SENDPOWEREVENT (40h) - Send Power Event

POWER_SENDPOWEREVENT (40h) - Description

This function enables the caller to initiate a Power Management event through the Power Management Services subsystem and system clients.

POWER_SENDPOWEREVENT (40h) - ulParm1

ulParm1

The *ulParm1* field is used as defined for the Power Management event messages.

POWER_SENDPOWEREVENT (40h) - ulParm2

ulParm2

The *ulParm2* field is used as defined for the Power Management event messages.

OS/2 Power Management Event Messages	ulParm1 SubID	ulParm1 Reserved	ulParm2 (msg-specific)
Enable Pwr Mgt Functions	0003h	0000h	- - - - - - - -

Disable Pwr Mgt Functions	0004h	0000h	- - -	- - -
Restore BIOS Defaults	0005h	0000h	- - -	- - -
Set Power State	0006h	0000h	DevID	PwrState
Battery Low Event	0007h	0000h	- - -	- - -
Normal Resume Event	0008h	0000h	- - -	- - -
Critical Resume Event	0009h	0000h	- - -	- - -

Note: All other values are reserved.

Power Management Event Messages and Parameter Values

POWER_SENDPOWEREVENT (40h) - Parameter Packet Format

All parameter packet fields are input fields for this function.

Field	Length	C Datatype
ulParm1	DWORD	ULONG
ulParm2	DWORD	ULONG

POWER_SENDPOWEREVENT (40h) - ReturnCode

ReturnCode

The function-specific completion value. The valid return values are:

PowerNoError	(0000H)
PowerBadSubId	(0001H)
PowerBadReserved	(0002H)
PowerBadDevId	(0003H)
PowerBadPwrState	(0004H)
PowerDisabled	(0009H)

POWER_SENDPOWEREVENT (40h) - Data Packet Format

This data packet field is an output field for this function.

Field	Length	C Datatype
ReturnCode	WORD	USHORT

POWER_SENDPOWEREVENT (40h) - Returns

The *ReturnCode* field is in the Data Packet.

When an error condition is encountered with an input packet field (Parameter), the INVALID_PARAMETER IOCtl error code (13H) is returned to the caller. The caller must then inspect the Data Packet's *ReturnCode* field to determine the specific nature of the failure.

POWER_SENDPOWEREVENT (40h) - Remarks

The *SubId* values permitted in the *ulParm1* field are limited to values 03H through 06H. If the caller requests a SubId value outside this supported range, the request returns with a PowerBadSubId error code.

When Power Management is disabled (SubId=0004H), all requests except Power Management Enable (SubId=0003H) fail and are returned to the caller with a ReturnCode value of PowerDisabled.

POWER_SENDPOWEREVENT (40h) -

Category:
IOCTL_POWER (0Ch)
Function:
POWER_SENDPOWEREVENT (40h)
Description:
Send Power Event

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

POWER_SETPOWEREVENTRES (41h) - Set Power Event Resour

POWER_SETPOWEREVENTRES (41h) - Description

This function enables the caller to manipulate the resource set used to communicate with the Power Management subsystem. The Parameter Packet fields enable the caller to customize the resource set to fit the communication protocol desired.

POWER_SETPOWEREVENTRES (41h) - ReqFlags

ReqFlags
A bit mask with the following bit definitions:

Bit #	Meaning If Set
0	SemSetup request
1	SemRemoval request
2	16BitSemFlag
3	32BitSemFlag
4	NoSemFlag

All other bits are reserved and must be 0.

If any of the reserved bits in the *ReqFlags* field are set (value = 1), the request fails with a ReturnCode value of PowerBadReserved.

SemSetup requests (bit 0 set) must have the appropriate semaphore type defined (bit 2 or 3 set). SemSetup requests that do not have the semaphore type properly defined (bit 2 or 3 set, but not both) are returned to the caller with a ReturnCode value of PowerBadFlags.

Requests that are not manipulating the semaphore resource must have the NoSemFlag (bit 4) set and all other bits clear. Requests that do no conform to this requirement are returned to the caller with a ReturnCode value of PowerBadFlags.

POWER_SETPOWEREVENTRES (41h) - SemHandle

SemHandle

A semaphore handle created for Power Management Event Notifications. This function supports the use of 16-bit system semaphores or 32-bit event semaphores. When a semaphore is added to the resource set, the appropriate bits must be specified in the *ReqFlags* field to define the semaphore type. This field value is not required if the caller's request is not involved in setup or removal of a semaphore resource.

Note:

If using a 16-bit semaphore, you must use a 16-bit system semaphore. RAM semaphores are not supported.

If using a 32-bit event semaphore, the resource must be created with the *Shared* attribute so the Power Management subsystem can access it at any time.

POWER_SETPOWEREVENTRES (41h) - EventMask

EventMask

A bit mask that defines the specific event messages permitted to be written into the caller's event queue. The following bits are defined for the *EventMask* field:

Bit #	Meaning If Set
3	Enable Power Management
4	Disable Power Management
5	Restore BIOS Defaults
6	Set Power State
7	Battery Low Event
8	Normal Resume Event
9	Critical Resume Event All other bits are reserved and must be 0.

If any of the reserved bits in the *EventMask* field are set (value = 1), the request fails with a ReturnCode value of PowerBadReserved.

POWER_SETPOWEREVENTRES (41h) - Parameter Packet Forma

All parameter packet fields are input fields for this function.

Field	Length	C Datatype
ReqFlags	WORD	USHORT
SemHandle	DWORD	ULONG
EventMask	DWORD	ULONG

POWER_SETPOWEREVENTRES (41h) - ReturnCode

ReturnCode

The function-specific completion value. The valid return values are:

PowerNoError	(0000H)
PowerBadReserved	(0002H)
PowerSemAlreadySetup	(0005H)
PowerBadFlags	(0006H)
PowerBadSemHandle	(0007H)
PowerDisabled	(0009H)
PowerTooManyQueues	(000BH)
PowerBadSemaphore	(000DH)

POWER_SETPOWEREVENTRES (41h) - Data Packet Format

All data packet fields are output fields for this function.

Field	Length	C Datatype
ReturnCode	WORD	USHORT

POWER_SETPOWEREVENTRES (41h) - Returns

The *ReturnCode* field is in the Data Packet.

When an error condition is encountered with an input packet field (Parameter), the INVALID_PARAMETER IOCtl error code (13H) is returned to the caller. The caller must then inspect the Data Packet's *ReturnCode* field to determine the specific nature of the failure.

POWER_SETPOWEREVENTRES (41h) - Remarks

A set of Power Management resources is allocated when an application calls this IOCtl. This resource set consists of a queue that holds Power Management event messages. Only one resource set (event queue) can be created per file handle.

If a semaphore is established, the caller is notified (that is, the semaphore is cleared) each time a Power Management event message is added to the queue. It is the responsibility of the calling application to reset the semaphore. Because different applications might be interested in different event messages, *EventMask* can be tailored to specify only those event messages the application is interested in processing. The resource set is freed when DosClose is performed on the associated file handle.

If the resource set is defined without a semaphore, the IOCtl 61h interface must be polled to determine whether any events have occurred (that is, if any event messages are in the queue).

If a SemRemoval request (Bit 1 is set) attempts to release a semaphore with a bad handle, or to release a handle to a semaphore that is different from the handle provided in the SemSetup request (bit set to 0), then PowerBadSemHandle is referenced in the *ReturnCode* field.

When a semaphore handle has been provided, all subsequent Setup requests for that resource set fail with a ReturnCode value of PowerSemAlreadySetup unless a corresponding SemRemoval request has been processed.

If an input semaphore handle value is rejected by the DevHlp semaphore management functions, the request fails with a ReturnCode value of PowerBadSemaphore.

POWER_SETPOWEREVENTRES (41h) -

Category:
IOCTL_POWER (0Ch)
Function:
POWER_SETPOWEREVENTRES (41h)
Description:
Set Power Event Resource

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

POWER_GETPOWERSTATUS (60h) - Query Power Status

POWER_GETPOWERSTATUS (60h) - Description

This function returns the current power state information for the system and system devices.

POWER_GETPOWERSTATUS (60h) - ParmLength

ParmLength
Contains the size of the Parameter Packet in bytes (including *ParmLength*). On input, it must be a fixed value of 7 bytes.

POWER_GETPOWERSTATUS (60h) - PowerFlags

PowerFlags

A bit mask with the following bit definition:

Bit #	Meaning
0	Power Management State

All other bits are reserved and must be 0.

If the Power Management State (bit 0) is set (value=1), then Power Management support is currently enabled. If the Power Management State is reset (value=0), then Power Management support is currently disabled.

Note: The Power Management subsystem interfaces remain available even when the support has been disabled (by way of the Send Power Event IOCtl). However, the status data returned in the other Parameter Packet fields for this request might not contain meaningful data values.

POWER_GETPOWERSTATUS (60h) - ACStatus

ACStatus

Contains the status value for the AC line power source. Supported values are:

0	= AC Offline
1	= AC Online
255	= Unknown

All other values are reserved.

POWER_GETPOWERSTATUS (60h) - BatteryStatus

BatteryStatus

Contains the status value for the DC battery power source. Supported values are:

0	= High
1	= Low
2	= Critical
3	= Charging
255	= Unknown

All other values are reserved.

POWER_GETPOWERSTATUS (60h) - BatteryLife

BatteryLife

Contains the remaining power level for the DC battery power source. Supported values are:

0 to 100	= Percentage of battery life remaining
255	= Unknown

All other values are reserved.

POWER_GETPOWERSTATUS (60h) - Parameter Packet Format

The majority of the parameter packet fields are output fields for this function. The exception is the *ParmLength* field, which is both input and output.

Field	Length	C Datatype
ParmLength	WORD	USHORT
PowerFlags	WORD	USHORT
ACStatus	BYTE	UCHAR
BatteryStatus	BYTE	UCHAR
BatteryLife	BYTE	UCHAR

POWER_GETPOWERSTATUS (60h) - ReturnCode

ReturnCode

The function-specific completion value. The valid return values are:

PowerNoError	(0000H)
PowerBadLength	(0008H)
PowerDisabled	(0009H)

POWER_GETPOWERSTATUS (60h) - Data Packet Format

All data packet fields are output fields for this function.

Field	Length	C Datatype
ReturnCode	WORD	USHORT

POWER_GETPOWERSTATUS (60h) - Returns

Both Parameter and Data Packets contain return information.

When an error condition is encountered with an input packet field (Parameter), the INVALID_PARAMETER IOCtl error code (13H) is returned to the caller. The caller must then inspect the Data Packet's *ReturnCode* field to determine the specific nature of the failure.

POWER_GETPOWERSTATUS (60h) - Remarks

On input, the *ParmLength* field indicates the size of the Parameter Packet in bytes. On output, the *ParmLength* field indicates the minimum required size (in bytes) for the function.

POWER_GETPOWERSTATUS (60h) -

Category:
IOCTL_POWER (0Ch)
Function:
POWER_GETPOWERSTATUS (60h)
Description:
Query Power Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

POWER_GETPOWEREVENT (61h) - Query Power Event

POWER_GETPOWEREVENT (61h) - Description

This function returns Power Management Event information queued in the Power Management Services subsystem.

POWER_GETPOWEREVENT (61h) - ParmLength

ParmLength
Contains the size of the Parameter Packet in bytes (including *ParmLength*). On input, it must be a fixed value of 12 bytes.

POWER_GETPOWEREVENT (61h) - MsgCount

MsgCount
On output, indicates the number of remaining queue elements currently queued. On input, this field value is undefined.

POWER_GETPOWEREVENT (61h) - ulParm1

ulParm1

The *ulParm1* field is used as defined for the Power Management event messages.

POWER_GETPOWEREVENT (61h) - ulParm2

ulParm2

The *ulParm2* field is used as defined for the Power Management event messages.

POWER_GETPOWEREVENT (61h) - Parameter Packet Format

The majority of the parameter packet fields are output fields for this function. The exception is the *ParmLength* field, which is both input and output.

Field	Length	C Datatype
ParmLength	WORD	USHORT
MsgCount	WORD	USHORT
ulParm1	DWORD	ULONG
ulParm2	DWORD	ULONG

POWER_GETPOWEREVENT (61h) - ReturnCode

ReturnCode

The function-specific completion value. The valid return values are:

PowerNoError	(0000H)
PowerBadLength	(0008H)
PowerNoEventQueue	(000AH)
PowerQueueOverflow	(000EH)

POWER_GETPOWEREVENT (61h) - Data Packet Format

All Data Packet fields are output fields for this function.

Field	Length	C Datatype
ReturnCode	WORD	USHORT

POWER_GETPOWEREVENT (61h) - Returns

Both Parameter and Data Packets contain return information.

When an error condition is encountered with an input packet field (Parameter or Data) the INVALID_PARAMETER IOCtl error code (13H) is returned to the caller. The caller must then inspect the Data Packet's *ReturnCode* field to determine the specific nature of the failure.

POWER_GETPOWEREVENT (61h) - Remarks

On input, the *ParmLength* field indicates the size of the Parameter Packet in bytes. On output, the *ParmLength* field indicates either the number of bytes written (without error) or the minimum required size for the function (with the PowerBadLength error code value) to be completed.

POWER_GETPOWEREVENT (61h) -

Category:
IOCTL_POWER (0Ch)
Function:
POWER_GETPOWEREVENT (61h)
Description:
Query Power Event

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

POWER_GETPOWERINFO (62h) - Query Power Information

POWER_GETPOWERINFO (62h) - Description

This function returns the Power Management subsystem and BIOS configuration information.

POWER_GETPOWERINFO (62h) - ParmLength

ParmLength

ParmLength contains the size of the Parameter Packet in bytes (including *ParmLength*). On input, it must be a fixed value of 8 bytes.

POWER_GETPOWERINFO (62h) - BIOSFlags

BIOSFlags

BIOSFlags is a bit mask field with the bit definitions and setting as queried at system initialization. For specific bit definitions, consult the appropriate Advanced Power Management BIOS technical reference materials for the system unit.

POWER_GETPOWERINFO (62h) - BIOSVersion

BIOSVersion

BIOSVersion is a complex field composed of the following BYTE values:

- The high BYTE contains the Advanced Power Management BIOS major revision level value.
- The low BYTE contains the Advanced Power Management BIOS minor revision level value.

A combined WORD value of 0 indicates that no Advanced Power Management BIOS support is active or available. For the meaning of a specific version number, consult the appropriate BIOS technical reference materials for the system unit.

POWER_GETPOWERINFO (62h) - SubsysVersion

SubsysVersion

SubsysVersion is a complex field composed of the following BYTE values:

- The high BYTE contains the subsystem's major revision level value.
- The low BYTE contains the subsystem's minor revision level value.

For the first release of the Power Management subsystem, the Major revision level value is 1, and the Minor revision level is 0.

POWER_GETPOWERINFO (62h) - Parameter Packet Format

The majority of the parameter packet fields are output fields for this function. The exception is the *ParmLength* field, which is both input and output.

Field	Length	C Datatype
ParmLength	WORD	USHORT
BIOSFlags	WORD	USHORT
BIOSVersion	WORD	USHORT

SubsysVersion	WORD	USHORT

POWER_GETPOWERINFO (62h) - ReturnCode

ReturnCode

ReturnCode is the function-specific completion value. The valid return values are:

PowerNoError	(0000H)
PowerBadLength	(0008H)

POWER_GETPOWERINFO (62h) - Data Packet Format

All data packet fields are output fields for this function.

Field	Length	C Datatype
ReturnCode	WORD	USHORT

POWER_GETPOWERINFO (62h) - Returns

Both Parameter and Data Packets contain return information.

When an error condition is encountered with an input packet field (Parameter) the INVALID_PARAMETER IOCtl error code (13H) is returned to the caller. The caller must then inspect the Data Packet's *ReturnCode* field to determine the specific nature of the failure.

POWER_GETPOWERINFO (62h) - Remarks

On input, the *ParmLength* field indicates the size of the Parameter Packet in bytes. On output, the *ParmLength* field indicates the minimum required size (in bytes) for the function.

POWER_GETPOWERINFO (62h) -

Category:
IOCTL_POWER (0Ch)
Function:
POWER_GETPOWERINFO (62h)
Description:
Query Power Information

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 80h Screen Control IOCtl Commands

The following video IOCtls are defined and supported by the SCREENDD$ device driver, by way of the DosDevIOCtl call. The IOCtl category code is 80h (defined as SCREENDD_CATEGORY).

The function codes within the SCREENDD_CATEGORY are:

Function	Description
00h	Get Current Video Memory Bank
01h	Set Current Video Memory Bank
02h-07h	Reserved
08h	Return Adapter Video Configuration
09h	Return Manufacturer-Specific Adapter Data
0Ah	Update Adapter Video Memory Information
0Bh	Return Linear Address Mapped to Physical Address
0Ch-7Fh	Reserved

An example of the DosDevIOCtl calling convention for the Screen IOCtls follows:

```
int PASCAL near videoIoctl(VOID *data,VOID *parm,USHORT function)
{
  unsigned hScreenDD;                  /* handle of SCREENDD$ dev driver */
  unsigned OpenAction;                 /* action taken to open device    */
  unsigned rc;                         /* function return code           */


  if (!(rc = DosOpen(SCREENDD_NAME, (PHFILE)&hScreenDD, (PUSHORT)&OpenAction,
     NO_SIZE, NO_ATTRIBUTES, OPEN_IF_EXISTS, NO_INHERIT+DENY_NONE+READ_WRITE,
     RESERVED_LONG)))
  {
    rc = DosDevIOCtl(data,
                     parm,
                     function,
                     SCREENDD_CATEGORY,
                     (HFILE)hScreenDD);
    DosClose(hScreenDD);
  }
  return (rc);
```

SCREENDD_GETCURRENTBANK (00h) - Get Current Video Mem

SCREENDD_GETCURRENTBANK (00h) - Description

This function returns information about the current video memory bank.

SCREENDD_GETCURRENTBANK (00h) - Packet Length

Packet Length

Represents the combined length of all parameter packet fields. This is a required field for all calls, including those made by way of the 32-bit DosDevIOCtl, with a minimum packet length of 10 bytes. If the verification of the packet field fails, the ERROR_124_INVALID_PARAMETER message is returned.

SCREENDD_GETCURRENTBANK (00h) - Current Bank

Current Bank

The current bank value is returned in this field of the packet. The bank size is 64KB regardless of the value specified for the video mode type field.

SCREENDD_GETCURRENTBANK (00h) - Current Video Mode Typ

Current Video Mode Type

Defines the adapter video mode type to the device driver, and has one of the following values:

0	Text Mode
1	Planar Mode
2	Linear Mode

This field is required.

SCREENDD_GETCURRENTBANK (00h) - Read/Write Bank Mode

Read/Write Bank Mode

Specifies what bank type is to be returned, and has one of the following values:

0	Read
1	Write

This field is required.

SCREENDD_GETCURRENTBANK (00h) - Parameter Packet Form

Field	Length	C Datatype
Packet Length	DWORD	ULONG
Current Bank	WORD	USHORT
Current Video Mode Type	WORD	USHORT
Read/Write Bank Mode	WORD	USHORT

Related C Data Structure

The BANKINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

SCREENDD_GETCURRENTBANK (00h) - Data Packet Format

None.

SCREENDD_GETCURRENTBANK (00h) - Returns

None.

SCREENDD_GETCURRENTBANK (00h) - Remarks

This function returns the actual bank setting if the adapter is identified as one of the supported SVGA adapters. Refer to SCREENDD_SVGA_ID to determine the video adapter configuration. If this is not a supported SVGA adapter, the IOCtl returns 0 as the bank field value.

This function should be called by an application only while it owns the video display hardware (running in the foreground).

SCREENDD_GETCURRENTBANK (00h) -

Category:
SCREENDD_CATEGORY (80h)
Function:
SCREENDD_GETCURRENTBANK (00h)
Description:
Get Current Video Memory Bank

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCREENDD_SETCURRENTBANK (01h) - Set Current Video Memo

SCREENDD_SETCURRENTBANK (01h) - Description

This function sets the current video memory bank.

SCREENDD_SETCURRENTBANK (01h) - Packet Length

Packet Length

Represents the combined length of all parameter packet fields. This is a required field for all calls, including those made by way of the 32-bit DosDevIOCtl, with a minimum packet length of 10 bytes. If the verification of the packet field fails, the ERROR_124_INVALID_PARAMETER message is returned.

SCREENDD_SETCURRENTBANK (01h) - Current Bank

Current Bank

Defines the bank value to be set. The bank size is 64KB regardless of the value specified for the video mode type field.

SCREENDD_SETCURRENTBANK (01h) - Current Video Mode Typ

Current Video Mode Type

Defines the adapter video mode type to the device driver, and has one of the following values:

0	Text Mode
1	Planar Mode
2	Linear Mode

This field is required.

SCREENDD_SETCURRENTBANK (01h) - Read/Write Bank Mode

Read/Write Bank Mode

Specifies what bank type is to be set, and has one of the following values:

0	Read
1	Write

This field is required.

SCREENDD_SETCURRENTBANK (01h) - Parameter Packet Form

Field	Length	C Datatype
Packet Length	DWORD	ULONG
Current Bank	WORD	USHORT
Current Video Mode Type	WORD	USHORT
Read/Write Bank Mode	WORD	USHORT

Related C Data Structure

The BANKINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

SCREENDD_SETCURRENTBANK (01h) - Data Packet Format

None.

SCREENDD_SETCURRENTBANK (01h) - Returns

None.

SCREENDD_SETCURRENTBANK (01h) - Remarks

This function will set the bank only if the adapter is identified as one of the supported SVGA adapters. Refer to SCREENDD_SVGA_ID to determine the video adapter configuration. If this is not a supported SVGA adapter, the IOCtl returns a successful completion code without affecting the hardware.

This function should be called by an application only while it owns the video display hardware (running in the foreground).

SCREENDD_SETCURRENTBANK (01h) -

Category:
SCREENDD_CATEGORY (80h)
Function:
SCREENDD_SETCURRENTBANK (01h)
Description:
Set Current Video Memory Bank

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCREENDD_SVGA_ID (08h) - Return Adapter Video Configuration

SCREENDD_SVGA_ID (08h) - Description

This function returns the video adapter configuration.

SCREENDD_SVGA_ID (08h) - Parameter Packet Format

None.

SCREENDD_SVGA_ID (08h) - Adapter Type

Adapter Type
Returns the code value of the chip set manufacturer, and has one of the following values:

0	Indeterminate chip set
1	Headland Technology
2	Trident Microsystems
3	Tseng Laboratories
4	Western Digital
5	ATI Technologies
6	IBM
7	Cirrus Logic
8	S3 Incorporated
9	Chips and Technologies
10	WEITEK Corporation
11	Advance Logic

SCREENDD_SVGA_ID (08h) - Chip Type

Chip Type

Returns the chip type and has a value according to the specific manufacturer. Refer to the following table for more information.

Manufacturer	Chip set	Adapter type	Chip type
Indeterminate chip set		0	0
Headland Technology	HT205	1	1
	HT208	1	2
	HT209	1	3
Trident Microsystems	8800	2	1
	8900	2	2
Tseng Laboratories	ET3000	3	1
	ET4000	3	2
Western Digital	PVGA1A	4	1
	WD90C00	4	2
	WD90C11	4	3
	WD90C30	4	4
ATI Technologies	18800	5	1
	28800	5	2
IBM	VGA-256C	6	1
Cirrus Logic	GD5422	7	1
	GD5424	7	2
	GD5426	7	3

SCREENDD_SVGA_ID (08h) - Video Memory

Video Memory

Returns the detected memory size of the video adapter.

SCREENDD_SVGA_ID (08h) - Data Packet Format

Field	Length	C Datatype
Adapter Type	WORD	USHORT
Chip Type	WORD	USHORT
Video Memory	DWORD	ULONG

Related C Data Structure

The OEMSVGAINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

SCREENDD_SVGA_ID (08h) - Returns

None.

SCREENDD_SVGA_ID (08h) - Remarks

The screen device driver performs chip detection at initialization time and stores this information in an internal structure. When this IOCtl is called, the internal structure is copied into the data packet. This information is also available by calling OEMHLP$ device, category 80h, Function 08h. The OEMHLP device driver does not perform the actual detection but queries the physical screen device driver.

Future detection logic for new chip sets can be added to the screen device driver. Both the OEMHLP and Screen IOCtls will return these new values. If an OEM plans to include support for a new chip, the screen device driver's detection logic must be extended. Chip set values from 0-2FFh and FFF0-FFFF are reserved. Chip sets with a value of zero are not supported by OS/2 2.1. This does not imply that OS/2 will not operate or that the operating system will not take advantage of the video adapter installed.

Adapters type 9 (Chips and Technologies) and 11 (Advance Logic) are detected but not supported by IBM in non-VGA modes.

An application can update the video memory information returned in the data packet. However, this update does not affect the screen physical driver's internal configuration or processing. Refer to SCREENDD_UPDATEMEMORY for this function.

Only applications making Function 08h calls after the update receive the revised video memory value.

SCREENDD_SVGA_ID (08h) -

Category:
SCREENDD_CATEGORY (80h)
Function:
SCREENDD_SVGA_ID (08h)
Description:
Return Adapter Video Configuration

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCREENDD_SVGA_OEM (09h) - Return Manufacturer-Specific Ada

SCREENDD_SVGA_OEM (09h) - Description

This function returns identification data specific to a video board manufacturer.

SCREENDD_SVGA_OEM (09h) - Parameter Packet Format

None.

SCREENDD_SVGA_OEM (09h) - Packet Length

Packet Length

Represents the combined length of all parameter packet fields. This is a required field for all calls, including those made by way of the 32-bit DosDevIOCtl, with a minimum packet length of 10 bytes. If the verification of the packet field fails, the ERROR_124_INVALID_PARAMETER message is returned.

SCREENDD_SVGA_OEM (09h) - Manufacturer Name ID

Manufacturer Name ID

Identifies the name of the manufacturer, and currently has one of the following values:

Manufacturer	Product	Value
Diamond Computer Systems, Inc.	SpeedStar**, 24 Line	1
Orchid Technology, Inc.	Prodesigner** Series	2

.

SCREENDD_SVGA_OEM (09h) - Manufacturer Specific Data

Manufacturer Specific Data

Manufacturer-specific data is in a format designated by the manufacturer. For Diamond Computer Systems, values returned are BIOS major value (high WORD) and BIOS minor value (low WORD). For Orchid Technology, the value returned represents the Micro Channel slot number.

SCREENDD_SVGA_OEM (09h) - Data Packet Format

Field	Length	C Datatype
Packet Length	DWORD	ULONG
Manufacturer Name ID	WORD	USHORT
Manufacturer Specific Data	DWORD	ULONG

Related C Data Structure

The OEMINFO data structure can be used by C programmers.

To include this data structure in your file, make sure INCL_DOSDEVIOCTL is defined before you include OS2.H.

SCREENDD_SVGA_OEM (09h) - Returns

None.

SCREENDD_SVGA_OEM (09h) - Remarks

In some instances, it might be necessary to provide additional data to applications and the system that is unique to a specific adapter. This function facilitates this processing.

The manufacturer-specific data field is open to future expansion.

If the Manufacturer Name ID returned is a zero, the identification for this manufacturer is not currently implemented. This does not imply that the adapter is not supported. Special identification is provided for the adapters in addition to SVGA that require special handling.

SCREENDD_SVGA_OEM (09h) -

Category:
IOCTL_SCREEN (80h)
Function:
SCREENDD_SVGA_OEM (09h)
Description:
Return Manufacturer-Specific Adapter Data

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCREENDD_UPDATEMEMORY (0Ah) - Update Adapter Video Me

SCREENDD_UPDATEMEMORY (0Ah) - Description

Allows the update of internal video configuration structures and memory data.

SCREENDD_UPDATEMEMORY (0Ah) - New Video Memory Size

New Video Memory Size
Size of memory.

SCREENDD_UPDATEMEMORY (0Ah) - Parameter Packet Format

Field	Length	C Datatype
New Video Memory Size	DWORD	ULONG

SCREENDD_UPDATEMEMORY (0Ah) - Data Packet Format

None.

SCREENDD_UPDATEMEMORY (0Ah) - Returns

None.

SCREENDD_UPDATEMEMORY (0Ah) - Remarks

The screen physical device driver cannot accurately identify the size of the video memory for all adapters. This is because all of the necessary information for some of the adapters cannot be obtained in protect mode. In order to solve this problem, the Super VGA base video handler updates its memory information by calling the IOCtl. Information supplied to the IOCtl is obtained from the SVGADATA.PMI file.

If you run applications that use this IOCtl to modify the video memory size, be aware that there is no mechanism for identifying changes to other applications that previously used Function 08h to obtain video memory size.

SCREENDD_UPDATEMEMORY (0Ah) -

Category:
SCREENDD_CATEGORY (80h)
Function:

SCREENDD_UPDATEMEMORY (0Ah)

Description:

Update Adapter Video Memory Information

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

SCREENDD_GETLINEARACCESS (0Bh) - Return Linear Address | Address

SCREENDD_GETLINEARACCESS (0Bh) - Description

Returns the linear address mapped to a given physical address.

SCREENDD_GETLINEARACCESS (0Bh) - Packet Length

Packet Length

Represents the combined length of all parameter packet fields. This is a required field for all calls, including those made by way of the 32-bit DosDevIOCtl, with a minimum packet length of 10 bytes. If the verification of the packet field fails, the ERROR_124_INVALID_PARAMETER message is returned.

SCREENDD_GETLINEARACCESS (0Bh) - Physical Address

Physical Address

Represents the physical address where the CPU addresses the video memory of the video adapter. This is also called the video aperture or the CPU video memory window.

SCREENDD_GETLINEARACCESS (0Bh) - Aperture Size

Aperture Size

Represents the size of the video memory, which can be accessed at the window starting at the physical address.

SCREENDD_GETLINEARACCESS (0Bh) - Linear Address

Linear Address

Returns the linear address to which the physical address is mapped. Protect mode applications must use this linear address to access video memory.

SCREENDD_GETLINEARACCESS (0Bh) - Linear Flags

Linear Flags

Valid flags are defined in the following list:

```
GETLINEAR_FLAG_MAPPHYSICAL    0x00000010L
GETLINEAR_FLAG_MAPPROCESS      0x00000020L
GETLINEAR_FLAG_MAPSHARED      0x00000400L
GETLINEAR_FLAG_MAPATTACH      0x80000000L
```

SCREENDD_GETLINEARACCESS (0Bh) - Parameter Packet Form

Field	Length	C Datatype
Packet Length	DWORD	ULONG
Physical Address	DWORD	ULONG
Aperture Size	DWORD	ULONG
Linear Address	DWORD	ULONG
Linear Flags	DWORD	ULONG

SCREENDD_GETLINEARACCESS (0Bh) - Data Packet Format

None.

SCREENDD_GETLINEARACCESS (0Bh) - Returns

None.

SCREENDD_GETLINEARACCESS (0Bh) - Remarks

If an application requires access to a video aperture other than 64KB between A000 and BFFF, it may obtain a linear address by way of this

IOCtl. The address is returned in the linear address field of the packet. The physical address and the aperture size are limited by the adapter's memory-mapping capabilities on a given system.

SCREENDD_GETLINEARACCESS (0Bh) -

Category:
SCREENDD_CATEGORY (80h)
Function:
SCREENDD_GETLINEARACCESS (0Bh)
Description:
Return Linear Address Mapped to Physical Address

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 80h OEMHLP IOCtls

The OEMHLP interface was originally designed to allow Original Equipment Manufacturers (OEMs) to modify and adapt the OS/2 operating system to run on their hardware. In the past, IBM supported the OS/2 operating system on IBM hardware only. Therefore, OEMs had to build modified versions of the OS/2 operating system. The OEMHLP interface facilitated this process.

IBM currently tests the OS/2 operating system on a wide variety of OEM hardware. It is no longer necessary for OEMs to adapt the OS/2 operating system to their machines. Now the OEMHLP interface can be used to obtain real-mode information. This information can be passed to applications and device drivers running in protect mode. Applications and physical device drivers running in protect mode cannot access BIOS through the INT interface. The OEMHLP interface allows access to BIOS information and functions that are essential to these programs.

For example, you might want to issue INT 15h calls from your device driver initialization code to determine if an Extended Industry Standard Architecture (EISA) adapter is present. The following examples show the methods to determine if a specific EISA or Micro Channel adapter is present.

Using the Query Adapter ID to Verify EISA Adapter

The following example uses the OEMHLP IOCtl interface to verify the EISA card ID:

```
USHORT FindMyEISACard(void)
{
  HFILE filehandle;
  USHORT action;

  EISAFunctionInfo.efi_SubFunc = OEM_GET_SLOT_INFO; /* EISA Get Slot Info */
  EISAFunctionInfo.efi_Slot    = 0;                 /* Slot 0             */

  rc = DosOpen("OEMHLP$",
               &filehandle,
               &action,
               0L,
               0,
               1,
               0x40,
               0L);
```

```
  if (rc == 0)
    {
    for(index=1;index<CFG_MAX_EISA_SLOTS;index++)   /* For each slot      */
      {
      EISAFunctionInfo.efi_Slot    = (UCHAR) index; /* Slot Number        */
      EISASlotInfo.esi_CardID  = 0;                 /* Reset Card ID value*/
      rc = DosDevIOCtl((PVOID)&EISASlotInfo,        /* Data Packet */
                       (PVOID)&EISAFunctionInfo,    /* Parm Packet */
                       (USHORT)OEMHLP_QUERYEISACONFIG,
                       (USHORT)OEMHLP_CATEGORY,
                       (HFILE)filehandle);
      /* If IOCtl successful and slot has adapter, then store away
         the adapter ID, otherwise mark as empty with a zero.
       */
      if((rc==0)&&(EISASlotInfo.esi_Error==0))
        {
        if (EISASlotInfo.esi_CardID == MYCARDID)
           DosClose(filehandle);        /* Close handle to OEMHLP$ */
           return(FOUND);
        }
      }
      DosClose(filehandle);             /* Close handle to OEMHLP$ *    /
    }
   return(NOTFOUND);
}
```

Using the DevHlp_ABIOSCall to Verify Micro Channel Adapter

The following example uses the DevHlp_ABIOSCall to verify the Micro Channel POS ID:

```
USHORT FindMyMicroChannelCard(void)
{
  USHORT i,rc;              /* Index and return code    */
  USHORT LID;               /* Logical ID               */

  if (GetLIDEntry(POS,0,1,&LID))
     return(NOTFOUND);

  /* Get length of RB to use for reading POS data. */

  POSLenRB.rb.RBLen = sizeof(POSLENRB);
  POSLenRB.rb.Func  = 0x01;
  POSLenRB.rb.LID   = LID;
  POSLenRB.rb.Unit  = 0;
  POSLenRB.rb.Resv1 = 0;
  POSLenRB.rb.Resv2 = 0;
  POSLenRB.Rsv1     = 0;
  POSLenRB.Rsv2     = 0;
  POSLenRB.Rsv3     = 0;
  rc = ABIOSCall( LID, &POSLenRB, 0);

  /*Is my request block big enough? */

  if ((rc==0) && (sizeof(POSRB) >= POSLenRB.RBLen))
     {
       RB.rb.RBLen = POSLenRB.RBLen;       /* request block length        */
       RB.rb.Func  = 0x0b;                 /* read stored POS data to mem */
       RB.rb.LID   = LID;                  /* Logical ID                  */
       RB.rb.Unit  = 0;
       RB.DataBuf  = (ULONG)(FARPOINTER)&POSData;

       for(i=0;i<=CFG_MAX_POS_SLOTS;i++)   /* For each slot, get POS ID   */
          {
           RB.Slot = (UCHAR)i;
```

```
          rc = ABIOSCall(LID,&RB,0);
          if((rc==0)&&(RB.rb.RetCode==0))
            if (RB.AdapterID == MYCARD)
                {
                  FreeLIDEntry(LID);
                  return(FOUND);
                }
          }
     }

  FreeLIDEntry(LID);             /* Release LID Entry */
  return(NOTFOUND);
}
```

OEMHLP IOCtls Summary

The following is a summary of Category 80h OEMHLP IOCtl Commands:

Function	Description
00h	Query OEM Adaptation Information
01h	Query Machine Information
02h	Query Display Combination Code
03h	Return Video Fonts
04h	Read EISA Slot Configuration Information - Subfunction 00
04h	Read EISA Function Configuration Information - Subfunction 01
05h	Query ROM BIOS Information
06h	Query Miscellaneous Video Information
07h	Query Video Adapter
08h	Query SVGA Information
09h	Query Memory Information
0Ah	Query Display Mode, Query and Set (DMQS) Information
0Bh	Access PCI BIOS Information
0Bh	Query PCI BIOS Information - Subfunction 00h
0Bh	Find PCI Device - Subfunction 01h
0Bh	Find PCI Class Code - Subfunction 02h
0Bh	Read PCI Configuration Space - Subfunction 03h
0Bh	Write PCI Configuration Space - Subfunction 04h

OEMHLP_GETOEMADAPTIONINFO (00h) - Query OEM Adaptatio

OEMHLP_GETOEMADAPTIONINFO (00h) - Description

This function returns information about a specific OEM adaptation of the OS/2 operating system.

OEMHLP_GETOEMADAPTIONINFO (00h) - Parameter Packet For

None.

OEMHLP_GETOEMADAPTIONINFO (00h) - OEM Name

OEM Name
Contains the name of the OEM stored as an ASCIIZ string. Use this field when this is not an IBM version of the OS/2 operating system and you have added additional OEMHLP functions to the system.

OEMHLP_GETOEMADAPTIONINFO (00h) - Internal OS/2 Revisior

Internal OS/2 Revision
The OS/2 version number stored as an ASCIIZ string.

OEMHLP_GETOEMADAPTIONINFO (00h) - Data Packet Format

Field	Length	C Datatype
OEM Name	20 BYTES	UCHAR[20]
Internal OS/2 Revision	10 BYTES	UCHAR[10]

OEMHLP_GETOEMADAPTIONINFO (00h) - Returns

Both Parameter and Data Packets contain return information. Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETOEMADAPTIONINFO (00h) - Remarks

OEMs can add nonstandard OEMHLP IOCtls to the OS/2 operating system if they market the OS/2 operating system with their own product name. Programs that use these IOCtls work only with that OEM's adaptation of the OS/2 operating system. Before issuing nonstandard calls, your device driver initialization code should issue the Query OEM Adaptation Information IOCtl routine to verify the OEM name field.

OEMHLP_GETOEMADAPTIONINFO (00h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETOEMADAPTIONINFO (00h)
Description:
Query OEM Adaptation Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETMACHINEINFO (01h) - Query Machine Information

OEMHLP_GETMACHINEINFO (01h) - Description

This function returns hardware-specific information about the computer on which the OS/2 operating system is installed.

OEMHLP_GETMACHINEINFO (01h) - Parameter Packet Format

None.

OEMHLP_GETMACHINEINFO (01h) - Name of Manufacturer

Name of Manufacturer
Name of manufacturer from ROM (if available), stored as an ASCIIZ string.

OEMHLP_GETMACHINEINFO (01h) - Model Number

Model Number
Model number from ROM (if available), stored as an ASCIIZ string.

OEMHLP_GETMACHINEINFO (01h) - ROM Revision Number

ROM Revision Number
ROM revision number (if available), stored as an ASCIIZ string.

OEMHLP_GETMACHINEINFO (01h) - Data Packet Format

Field	Length	C Datatype
Name of Manufacturer	20 BYTES	UCHAR[20]
Model Number	10 BYTES	UCHAR[10]
ROM Revision Number	10 BYTES	UCHAR[10]

OEMHLP_GETMACHINEINFO (01h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL

163 ERROR_UNCERTAIN_MEDIA
165 ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETMACHINEINFO (01h) - Remarks

This function attempts to determine the name of the manufacturer, the computer model number, and the ROM revision number. If the computer cannot be identified, the fields that the IOCtl returns in the data packet are set to NULLs.

OEMHLP_GETMACHINEINFO (01h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETMACHINEINFO (01h)
Description:
Query Machine Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETDISPLAYCOMBCODE (02h) - Query Display Combi

OEMHLP_GETDISPLAYCOMBCODE (02h) - Description

This function returns the display combination code.

OEMHLP_GETDISPLAYCOMBCODE (02h) - Parameter Packet Fo

None.

OEMHLP_GETDISPLAYCOMBCODE (02h) - Display Combination

Display Combination Code
Code returned from INT 10h (AH=1Ah).

OEMHLP_GETDISPLAYCOMBCODE (02h) - Data Packet Format

Field	Length	C Datatype
Display Combination Code	BYTE	UCHAR

OEMHLP_GETDISPLAYCOMBCODE (02h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETDISPLAYCOMBCODE (02h) - Remarks

This function returns the display combination code through INT 10h (AH=1Ah). If the BIOS does not support the INT 10h function, the IOCtl returns a 0.

Refer to the *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference* or the technical reference for your personal computer for more information about these pointers.

OEMHLP_GETDISPLAYCOMBCODE (02h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETDISPLAYCOMBCODE (02h)
Description:
Query Display Combination Code

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETVIDEOFONTS (03h) - Return Video Fonts

OEMHLP_GETVIDEOFONTS (03h) - Description

This function returns pointers to the video fonts.

OEMHLP_GETVIDEOFONTS (03h) - Parameter Packet Format

None.

OEMHLP_GETVIDEOFONTS (03h) - 8 x 14 Font Pointer (segment:

8 x 14 Font Pointer (segment:offset)
Pointer to the 8 x 14 ROM font.

OEMHLP_GETVIDEOFONTS (03h) - 8 x 8 Font Pointer (segment:o

8 x 8 Font Pointer (segment:offset)
Pointer to the 8 x 8 ROM font.

OEMHLP_GETVIDEOFONTS (03h) - 8 x 8 Font (top) Pointer (segm

8 x 8 Font (top) Pointer (segment:offset)
Pointer to the 8 x 8 ROM font (top).

OEMHLP_GETVIDEOFONTS (03h) - 9 x 14 Font Pointer (segment:

9 x 14 Font Pointer (segment:offset)
Pointer to the 9 x 14 ROM font.

OEMHLP_GETVIDEOFONTS (03h) - 8 x 16 Font Pointer (segment:

8 x 16 Font Pointer (segment:offset)
Pointer to the 8 x 16 ROM font.

OEMHLP_GETVIDEOFONTS (03h) - 9 x 16 Font Pointer (segment:

9 x 16 Font Pointer (segment:offset)
Pointer to the 9 x 16 ROM font.

OEMHLP_GETVIDEOFONTS (03h) - Data Packet Format

Field	Length	C Datatype
8 x 14 Font Pointer (segment:offset)	DWORD	ULONG
8 x 8 Font Pointer (segment:offset)	DWORD	ULONG
8 x 8 Font (top) Pointer (segment:offset)	DWORD	ULONG
9 x 14 Font Pointer (segment:offset)	DWORD	ULONG
8 x 16 Font Pointer (segment:offset)	DWORD	ULONG
9 x 16 Font Pointer (segment:offset)	DWORD	ULONG

OEMHLP_GETVIDEOFONTS (03h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA

OEMHLP_GETVIDEOFONTS (03h) - Remarks

This function returns pointers to the video fonts through INT 10h (AH=1130h).

Pointers returned by this IOCtl are addressed running in real mode. These pointers must then be converted to addresses running in protected mode before the applications and device drivers can access them.

Refer to the *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference* or the technical reference for your personal computer for more information about these pointers.

OEMHLP_GETVIDEOFONTS (03h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETVIDEOFONTS (03h)
Description:
Return Video Fonts

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_READEISACONFIGINFO (04h) - Read EISA Slot Configu Subfunction 00

OEMHLP_READEISACONFIGINFO (04h) - Description

This function routes select function calls to the BIOS on Extended Industry Standard Architecture (EISA) hardware that conforms to the EISA specifications.

The Query EISA Configuration Information function supports two subfunctions. Each of these two subfunctions corresponds to an equivalent EISA BIOS INT 15h, AH=D8h function call. These two subfunctions are:

Subfunction Number	Description	Corresponding EISA BIOS Call
00	Read EISA Slot Configuration Information	INT 15, AH=D8, AL=00
01	Read EISA Function Configuration Information	INT 15, AH=D8, AL=01

OEMHLP_READEISACONFIGINFO (04h) - Subfunction Number

Subfunction Number

The Read EISA Configuration Information subfunction number as described above and in the following text:

00	Read EISA Slot Configuration Information
01	Read EISA Function Configuration Information

OEMHLP_READEISACONFIGINFO (04h) - Slot Number

Slot Number

The EISA slot number on the system board (planar slot=0).

OEMHLP_READEISACONFIGINFO (04h) - Function Number

Function Number

For the Read EISA Function Configuration Information (subfunction 01) only, this is the sub-subfunction to read. Refer to the technical reference documentation for your personal computer hardware for details on the meaning of this parameter.

OEMHLP_READEISACONFIGINFO (04h) - Parameter Packet Form

The Parameter Packet format is the same for its two subfunctions. The common Parameter Packet format is as follows:

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR
Slot Number	BYTE	UCHAR
Function Number	BYTE	UCHAR

OEMHLP_READEISACONFIGINFO (04h) - Data Packet Format

The returned Data Packet format varies according to the particular subfunction being executed. The Data Packet format for each of the two subfunctions is described separately.

The Data Packet format for the data returned by the Read EISA Slot Configuration Information (subfunction 00) follows. The Register column in the table indicates the registers that contain the data if the EISA BIOS was accessed directly, and not through OEMHLP.

Field	Length	Registers	C Datatype
Return Code	BYTE	AH	UCHAR
Return Code Flags	BYTE	AL	UCHAR
Major Revision level of configuration	BYTE	BH	UCHAR
Minor Revision level of configuration	BYTE	BL	UCHAR
Checksum (LSBYTE) of configuration file	BYTE	CL	UCHAR
Checksum (MSBYTE) of configuration file	BYTE	CH	UCHAR
Number of device functions	BYTE	DH	UCHAR
Combined function information byte	BYTE	DL	UCHAR
Four byte compressed ID	DWORD	DI and SI	ULONG

OEMHLP_READEISACONFIGINFO (04h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_READEISACONFIGINFO (04h) - Remarks

Refer to the technical reference documentation for your personal computer hardware for additional information on these returned Data Packet fields.

OEMHLP_READEISACONFIGINFO (04h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_READEISACONFIGINFO (04h)
Description:
Read EISA Slot Configuration Information - Subfunction 00

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Read EISA Function Configuration Information - Subfunction 01 - Fu

Category 80h, Function 04h - Description

This function routes select function calls to the BIOS on Extended Industry Standard Architecture (EISA) hardware that conforms to the EISA specifications.

The Query EISA Configuration Information function supports two subfunctions. Each of these two subfunctions corresponds to an equivalent EISA BIOS INT 15h, AH=D8h function call. These two subfunctions are:

Subfunction Number	Description	Corresponding EISA BIOS Call
00	Read EISA Slot Configuration Information	INT 15, AH=D8, AL=00
01	Read EISA Function Configuration Information	INT 15, AH=D8, AL=01

Category 80h, Function 04h - Subfunction Number

Subfunction Number

The Read EISA Configuration Information subfunction number as described above and in the following text:

00	Read EISA Slot Configuration Information
01	Read EISA Function Configuration Information

Category 80h, Function 04h - Slot Number

Slot Number

The EISA slot number on the system board (planar slot=0).

Category 80h, Function 04h - Function Number

Function Number

For the Read EISA Function Configuration Information (subfunction 01) only, this is the sub-subfunction to read. Refer to the technical reference documentation for your personal computer hardware for details on the meaning of this parameter.

Category 80h, Function 04h - Parameter Packet Format

The Parameter Packet format is the same for its two subfunctions. The common Parameter Packet format is as follows:

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR
Slot Number	BYTE	UCHAR
Function Number	BYTE	UCHAR

Category 80h, Function 04h - Data Packet Format

The returned Data Packet format varies according to the particular subfunction being executed. The Data Packet format for each of the two subfunctions is described separately.

Category 80h, Function 04h - Remarks

Various types and categories of data are returned by the Read Function Configuration Information call. The information returned includes the following data types:

- Return code
- Return code flags
- Four-byte Compressed ID
- Memory Configuration
- ID and Slot Information
- Interrupt Configuration
- CFG File Extension Revision Level
- DMA Channel Description
- Function Information
- Port I/O Information
- Type and Subtype ASCII String
- Initialization Data

The data returned by the Read Function Configuration Information call includes an EISA-defined 320-byte block. The format of this EISA-defined 320-byte block is dependent on both the EISA revision level of the hardware installed on your personal computer and the value of the returned Function Information byte. Refer to the technical reference documentation for your personal computer hardware for details of this 320-byte block. The Register column in the table indicates the register that contains the data if the EISA BIOS was accessed directly and not through OEMHLP.

Category 80h, Function 04h -

Category:

80h
Function:
04h
Description:
Read EISA Function Configuration Information - Subfunction 01

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

OEMHLP_GETROMBIOSINFO (05h) - Query ROM BIOS Informatic

OEMHLP_GETROMBIOSINFO (05h) - Description

This function returns ROM BIOS information.

OEMHLP_GETROMBIOSINFO (05h) - Parameter Packet Format

None.

OEMHLP_GETROMBIOSINFO (05h) - Model

Model
Machine model byte.

OEMHLP_GETROMBIOSINFO (05h) - Submodel

Submodel
Machine submodel byte.

OEMHLP_GETROMBIOSINFO (05h) - BIOS Revision Level

BIOS Revision Level
BIOS revision level.

OEMHLP_GETROMBIOSINFO (05h) - BIOS/ABIOS Flags

BIOS/ABIOS Flags

BIT 0	A value of 1 indicates that ABIOS is present.
BIT 1-15	Reserved.

OEMHLP_GETROMBIOSINFO (05h) - Data Packet Format

Field	Length	C Datatype
Model	WORD	USHORT
Submodel	WORD	USHORT
BIOS Revision Level	WORD	USHORT
BIOS/ABIOS Flags	WORD	USHORT

OEMHLP_GETROMBIOSINFO (05h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETROMBIOSINFO (05h) - Remarks

OS/2 2.0 does not support RAM-loaded ABIOS computers. OS/2 2.0 returns BIT 0 set to 0 on computers with RAM-loaded ABIOS.

OS/2 2.1 supports RAM-loaded ABIOS computers. OS/2 2.1 returns BIT 0 set to 1 on machines with RAM-loaded ABIOS.

The values placed in the model and submodel fields are byte values. The high-order byte of the model and submodel fields are 0.

OEMHLP_GETROMBIOSINFO (05h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETROMBIOSINFO (05h)
Description:
Query ROM BIOS Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETMISCVIDEOINFO (06h) - Query Miscellaneous Vide

OEMHLP_GETMISCVIDEOINFO (06h) - Description

This function returns miscellaneous video state information.

OEMHLP_GETMISCVIDEOINFO (06h) - Parameter Packet Format

None.

OEMHLP_GETMISCVIDEOINFO (06h) - Video State Information

Video State Information

Miscellaneous Video State Information:

Bit 7	Reserved.
Bit 6	P70 video adapter active.
Bit 5	Video attribute bit. =0 Background intensity. =1 Blinking.
Bit 4	Cursor emulation is active.
Bit 3	Mode set default. Palette loading is disabled.
Bit 2	Monochrome display is attached.
Bit 1	Summing capability is active.
Bit 0	All modes on all displays are active.

OEMHLP_GETMISCVIDEOINFO (06h) - Data Packet Format

Field	Length	C Datatype
Video State Information	BYTE	UCHAR

OEMHLP_GETMISCVIDEOINFO (06h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETMISCVIDEOINFO (06h) - Remarks

Bit 0 and Bit 4 are always set to 0 for the IBM PS/2 Model 8530.

Refer to the *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference* or the technical reference for your personal computer for more information about miscellaneous video state information.

OEMHLP_GETMISCVIDEOINFO (06h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETMISCVIDEOINFO (06h)
Description:
Query Miscellaneous Video Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETVIDEOADAPTER (07h) - Query Video Adapter

OEMHLP_GETVIDEOADAPTER (07h) - Description

This function returns the video adapter type.

OEMHLP_GETVIDEOADAPTER (07h) - Parameter Packet Format

None.

OEMHLP_GETVIDEOADAPTER (07h) - Video Adapter Type

Video Adapter Type

Bit 0	MPA
Bit 1	CGA
Bit 2	EGA
Bit 3	VGA
4-7	Reserved

OEMHLP_GETVIDEOADAPTER (07h) - Data Packet Format

Field	Length	C Datatype
Video Adapter Type	BYTE	UCHAR

OEMHLP_GETVIDEOADAPTER (07h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE

87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETVIDEOADAPTER (07h) - Remarks

Use Function 08h to determine SVGA video adapter information.

OEMHLP_GETVIDEOADAPTER (07h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETVIDEOADAPTER (07h)
Description:
Query Video Adapter

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETSVGAINFO (08h) - Query SVGA Information

OEMHLP_GETSVGAINFO (08h) - Description

This function returns SVGA video information.

OEMHLP_GETSVGAINFO (08h) - Parameter Packet Format

None.

OEMHLP_GETSVGAINFO (08h) - Adapter Type

Adapter Type

Returns the code value of the chip set manufacturer, and has one of the following values:

0	Indeterminate Chip Set
1	Headland Technology Inc.
2	Trident** Microsystems, Inc.
3	TSENG** Laboratories, Inc.
4	Western Digital** Corporation
5	ATI** Technologies, Inc.
6	IBM Corporation
7	Cirrus Logic**, Inc.
8	S3** Incorporated
10	Weitek** Corporation

OEMHLP_GETSVGAINFO (08h) - Chip Type

Chip Type

Returns the chip type and has a value of 1 through 4 according to the specific manufacturer. Refer to the following table for more information:

Manufacturer	Chip Set	Adapter Type	Chip Type
Indeterminate Chip Set		0	0
Headland Technology, Inc.	HT205	1	1
	HT208	1	2
	HT209	1	3
Trident Microsystems	8800	2	1
	8900	2	2
Tseng Labs	ET3000	3	1
	ET4000	3	2
	ET4000W32	3	3
	ET4000W32I	3	4
	ET4000W32IB	3	5
	ET4000W32IC	3	6
	ET4000W32PA	3	7
	ET4000W32PB	3	8
	ET4000W32PC	3	9
	ET4000W32ID	3	10
	ET4000W32PD	3	11
Western Digital	PVGA1A	4	1cx
	WD9000	4	2
	WD9011	4	3
	WD9030	4	4
	WD9026	4	5
	WD9027	4	6
	WD9031	4	7
	WD9024	4	8

	WD9033	4	9
ATI Technologies	18800	5	1
	28800	5	2
	38800	5	3
	68800	5	4
	88800	5	5
IBM	VGA-256C	6	1
	IBMSVGA	6	1
Cirrus Logic	5420	7	1
	5422	7	2
	5424	7	3
	5426	7	4
	5428	7	5
	5429	7	6
	543x	7	7
	5434	7	8
S3	86C805	8	1
	86C928	8	2
	86C911	8	3
	86C864	8	4
	86C964	8	5
Weitek	P9000	10	1
	W5186	10	2
	W5286	10	3

OEMHLP_GETSVGAINFO (08h) - Video Memory

Video Memory
Returns the detected memory size of the video adapter.

OEMHLP_GETSVGAINFO (08h) - Data Packet Format

Field	Length	C Datatype
Adapter Type	WORD	USHORT
Chip Type	WORD	USHORT

Video Memory	DWORD	ULONG

OEMHLP_GETSVGAINFO (08h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETSVGAINFO (08h) - Remarks

SVGA video information is also available by calling the screen IOCtls, Category 80h, Function 08h. Refer to the *Display Device Driver Reference* for information on these IOCtls.

OEMHLP_GETSVGAINFO (08h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETSVGAINFO (08h)
Description:
Query SVGA Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETMEMINFO (09h) - Query Memory Information

OEMHLP_GETMEMINFO (09h) - Description

This function returns the amount of random access memory available on the computer.

OEMHLP_GETMEMINFO (09h) - Parameter Packet Format

None.

OEMHLP_GETMEMINFO (09h) - Low Memory Size (kilobytes)

Low Memory Size (kilobytes)
The amount of random access memory below the 1 megabyte address. The value returned is in kilobytes.

OEMHLP_GETMEMINFO (09h) - High Memory Size (kilobytes)

High Memory Size (kilobytes)
The amount of random access memory above the 1 megabyte address. The value returned is in kilobytes.

In OS/2 2.1, the number of kilobytes in high memory is a ULONG field. Previous versions of the OS/2 operating system used a USHORT field. See *Remarks.*

OEMHLP_GETMEMINFO (09h) - Data Packet Format

Field	Length	C Datatype
Low Memory Size (kilobytes)	WORD	USHORT
High Memory Size (kilobytes)	DWORD	ULONG

OEMHLP_GETMEMINFO (09h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETMEMINFO (09h) - Remarks

In OS/2 2.1, the number of kilobytes in high memory is a DWORD field. Previous versions of the OS/2 operating system used a WORD field. Applications should query the version of the OS/2 operating system to determine the size of the data packet required. This query can be done by issuing an OEMHELP category 80h Function 00h or a DevHlp_GetDosVar with index=1, querying the MajorVersion and MinorVersion.

OEMHLP_GETMEMINFO (09h) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETMEMINFO (09h)
Description:
Query Memory Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_GETDMQSINFO (0Ah) - Query Display Mode Query and Information

OEMHLP_GETDMQSINFO (0Ah) - Description

This function returns a pointer to the XGA DMQS video information.

OEMHLP_GETDMQSINFO (0Ah) - Parameter Packet Format

None.

OEMHLP_GETDMQSINFO (0Ah) - DMQS Information Pointer (sele

DMQS Information Pointer (selector:offset)
Pointer to the XGA* Display Mode Query and Set (DMQS) information block.

OEMHLP_GETDMQSINFO (0Ah) - Data Packet Format

Field	Length	C Datatype
DMQS Information Pointer (selector:offset)	DWORD	ULONG

OEMHLP_GETDMQSINFO (0Ah) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
117	ERROR_INVALID_CATEGORY
119	ERROR_BAD_DRIVER_LEVEL
163	ERROR_UNCERTAIN_MEDIA
165	ERROR_MONITORS_NOT_SUPPORTED

OEMHLP_GETDMQSINFO (0Ah) - Remarks

The pointer returned is a protected-mode address. Applications and device drivers running in protected mode do not need to convert this address before using it.

The XGA DMQS information is available only for IBM XGA/2* adapters and compatible adapters.

Information about XGA Display Mode Query and Set (DMQS) can be found in the *IBM Personal System/2 Hardware Interface Technical Reference - Video Subsystem* .

OEMHLP_GETDMQSINFO (0Ah) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_GETDMQSINFO (0Ah)
Description:
Query Display Mode Query and Set (DMQS) Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

OEMHLP_PCI (0Bh) - Access PCI BIOS

OEMHLP_PCI (0Bh) - Description

This function routes select function calls to the BIOS on Peripheral Component Interconnect (PCI) hardware that conforms to the PCI BIOS 2.0 specification.

The Access PCI BIOS function supports five subfunctions. The subfunctions are as follows:

Subfunction Number	Description	Corresponding PCI BIOS Call
0	Query PCI BIOS Information	Int 1A AH=B1 AL=01
1	Find PCI Device	Int 1A AH=B1 AL=02
2	Find PCI Class Code	Int 1A AH=B1 AL=03
3	Read PCI Configuration Space	Int 1A AH=B1 AL=08, 09, or 0A
4	Write PCI Configuration Space	Int 1A AH=B1 AL=0B, 0C, or 0D

OEMHLP_PCI (0Bh) - Parameter Packet Format

The Parameter Packet format differs among each subfunction and each of the subfunctions are described separately.

OEMHLP_PCI (0Bh) - Data Packet Format

The Data Packet format differs among each subfunction and each of the subfunctions are described separately.

OEMHLP_PCI (0Bh) - Remarks

The Data Packets for each subfunction each contain a *byte* field for Return Code. This field contains the return code returned from their respective PCI BIOS call. These return codes are described as follows:

Return Code	**Description**
00h	Successful
81h	Function Not Supported
83h	Bad Vendor ID
86h	Device Not Found
87h	Bad Register Number

Further Reference can be found in the *PCI BIOS Specification Revision 2.0.*

OEMHLP_PCI (0Bh) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_PCI (0Bh)
Description:
Access PCI BIOS

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

OEMHLP_PCI (0Bh) - Query PCI BIOS Information - Subfunction 0(

OEMHLP_PCI (0Bh) - Description

This function returns information specific to the PCI BIOS installed on a machine, such as revision number and levels of support.

OEMHLP_PCI (0Bh) - Subfunction Number

Subfunction Number
The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

00	Query PCI BIOS Information

OEMHLP_PCI (0Bh) - Parameter Packet Format

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Return Code

Return Code

The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

00	Query PCI BIOS Information

OEMHLP_PCI (0Bh) - Hardware Mechanism

Hardware Mechanism

PCI BIOS hardware platform support.

BIT 0	Hardware Configuration Mechanism #1
BIT 1	Hardware Configuration Mechanism #2
BIT 2-3	Reserved (0)
BIT 4	Special Cycles supported. HW Mechanism #1
BIT 5	Special Cycles supported. HW Mechanism #2

OEMHLP_PCI (0Bh) - Major Version

Major Version

PCI BIOS Major Version Level.

OEMHLP_PCI (0Bh) - Minor Version

Minor Version

PCI BIOS Minor Version Level.

OEMHLP_PCI (0Bh) - Last Bus

Last Bus

Last PCI bus indexed sequentially within the system.

OEMHLP_PCI (0Bh) - Data Packet Format

Field	Length	C Datatype
Return Code	BYTE	UCHAR
Hardware Mechanism	BYTE	UCHAR
Major Version	BYTE	UCHAR
Minor Version	BYTE	UCHAR
Last Bus	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Remarks

None.

OEMHLP_PCI (0Bh) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_PCI (0Bh)
Description:
Query PCI BIOS Information - Subfunction 00h

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

OEMHLP_PCI (0Bh) - Find PCI Device - Subfunction 01h

OEMHLP_PCI (0Bh) - Description

Returns information on a device specified by vendor and device ID numbers.

OEMHLP_PCI (0Bh) - Subfunction Number

Subfunction Number

The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

01 Find PCI Device

OEMHLP_PCI (0Bh) - Device ID

Device ID

Device-specific (within Vendor ID) identification.

OEMHLP_PCI (0Bh) - Vendor ID

Vendor ID

Vendor-specific identification.

OEMHLP_PCI (0Bh) - Index

Index

Zero-based index to access identical devices on a system.

OEMHLP_PCI (0Bh) - Parameter Packet Format

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR
Device ID	WORD	USHORT
Vendor ID	WORD	USHORT
Index	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Return Code

Return Code

The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

01 Find PCI Device

OEMHLP_PCI (0Bh) - Bus Number

Bus Number

Bus location of PCI device.

OEMHLP_PCI (0Bh) - DevFunc Number

DevFunc Number

Device Number - Upper 5 bits. Function Number - lower 3 bits. Locates a PCI device on a specified bus.

OEMHLP_PCI (0Bh) - Data Packet Format

Field	Length	C Datatype
Return Code	BYTE	UCHAR
Bus Number	BYTE	UCHAR
DevFunc Number	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Remarks

To find identical PCI devices, successive calls must be made while incrementing the Index until a return code of 86h - *Device Not Found* is returned.

OEMHLP_PCI (0Bh) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_PCI (0Bh)
Description:
Find PCI Device - Subfunction 01h

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

OEMHLP_PCI (0Bh) - Find PCI Class Code - Subfunction 02h

OEMHLP_PCI (0Bh) - Description

Returns information on a device specified by Class Code.

OEMHLP_PCI (0Bh) - Subfunction Number

Subfunction Number

The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

02	Find PCI Class Code

OEMHLP_PCI (0Bh) - Class Code

Class Code

Code identifying a type of PCI device. Refer to the *PCI Specification* .

OEMHLP_PCI (0Bh) - Index

Index

Zero-based index to access identical devices on a system.

OEMHLP_PCI (0Bh) - Parameter Packet Format

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR
Class Code	DWORD	ULONG
Index	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Return Code

Return Code

The PCI BIOS return code as descibed under Function 0Bh - Access PCI BIOS.

02 Find PCI Class Code

OEMHLP_PCI (0Bh) - Bus Number

Bus Number

Bus location of PCI device.

OEMHLP_PCI (0Bh) - DevFunc Number

DevFunc Number

Device Number - Upper 5 bits. Function Number - lower 3 bits. Locates a PCI device on a specified bus.

OEMHLP_PCI (0Bh) - Data Packet Format

Field	Length	C Datatype
Return Code	BYTE	UCHAR
Bus Number	BYTE	UCHAR
DevFunc Number	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Remarks

To find identical PCI devices, successive calls must be made while incrementing the Index until a return code of 86h - *Device Not Found* is returned.

OEMHLP_PCI (0Bh) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_PCI (0Bh)
Description:
Find PCI Class Code - Subfunction 02h

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

OEMHLP_PCI (0Bh) - Read PCI Configuration Space - Subfunction

OEMHLP_PCI (0Bh) - Description

This function allows reading of a PCI Configuration register on a specified device.

OEMHLP_PCI (0Bh) - Subfunction Number

Subfunction Number

The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

03 Read PCI Configuration device.

OEMHLP_PCI (0Bh) - Bus Number

Bus Number

Bus location of PCI device.

OEMHLP_PCI (0Bh) - DevFuncNum

DevFuncNum

Device Number - Upper 5 bits. Function Number - lower 3 bits. Locates a PCI device on a specified bus.

OEMHLP_PCI (0Bh) - Configuration Register

Configuration Register

Configuration register to access.

OEMHLP_PCI (0Bh) - Size

Size

Specifies size of read. Allowable sizes are as follows:

01	BYTE
02	WORD
04	DWORD

OEMHLP_PCI (0Bh) - Parameter Packet Format

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR
Bus Number	BYTE	UCHAR
DevFuncNum	BYTE	UCHAR
Configuration Register	BYTE	UCHAR
Size	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Return Code

Return Code

The PCI BIOS as descibed under Function 0Bh - Access PCI BIOS.

03	Read PCI Configuration Space.

OEMHLP_PCI (0Bh) - Data

Data

Data read from configuration register.

OEMHLP_PCI (0Bh) - Data Packet Format

Field	Length	C Datatype
Return Code	BYTE	UCHAR
Data	DWORD	ULONG

OEMHLP_PCI (0Bh) - Remarks

Unused upper bytes of *Data* field will be zero-filled. Refer to the *PCI Specification* for more information on configuration registers.

OEMHLP_PCI (0Bh) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_PCI (0Bh)
Description:
Read PCI Configuration Space - Subfunction 03h

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

OEMHLP_PCI (0Bh) - Write PCI Configuration Space - Subfunction

OEMHLP_PCI (0Bh) - Description

Allows writing of a PCI Configuration register on a specified device.

OEMHLP_PCI (0Bh) - Subfunction Number

Subfunction Number

The Subfunction Number as descibed under Function 0Bh - Access PCI BIOS.

04	Write PCI Configuration device.

OEMHLP_PCI (0Bh) - Bus Number

Bus Number

Bus location of PCI device.

OEMHLP_PCI (0Bh) - DevFuncNum

DevFuncNum

Device Number - Upper 5 bits. Function Number - lower 3 bits. Locates a PCI device on a specified bus.

OEMHLP_PCI (0Bh) - Configuration Register

Configuration Register

Configuration register to access.

OEMHLP_PCI (0Bh) - Size

Size

Specifies size of write. Allowable sizes are as follows:

01	BYTE
02	WORD
04	DWORD

OEMHLP_PCI (0Bh) - Data

Data

Data to be written to PCI configuration register.

OEMHLP_PCI (0Bh) - Parameter Packet Format

Field	Length	C Datatype
Subfunction Number	BYTE	UCHAR
Bus Number	BYTE	UCHAR
DevFuncNum	BYTE	UCHAR
Configuration Register	BYTE	UCHAR
Size	BYTE	UCHAR
Data	DWORD	ULONG

OEMHLP_PCI (0Bh) - Data Packet Format

Field	Length	C Datatype
Return Code	BYTE	UCHAR

OEMHLP_PCI (0Bh) - Remarks

Unused upper bytes of the *Data* file will be ignored. Refer to the *PCI Specification* for more information on configuration registers.

OEMHLP_PCI (0Bh) -

Category:
IOCTL_OEMHLP (80h)
Function:
OEMHLP_PCI (0Bh)
Description:
Write PCI Configuration Space - Subfunction 04h

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

Category 80h Adapter Presence-Check Services (TESTCFG.SYS)

The TESTCFG device driver provides services for automatic detection of Original Equipment Manufacturer (OEM) hardware interfaces.
Functions provided by this driver are accessed entirely by opening the device name *TESTCFG$* and using the following Category 80h IOCtls.

Function	Description
40H	Get Copy of BIOS/Adapter Memory
41H	Issue an "IN" I/O Instruction
42H	Issue an "OUT" I/O Instruction
60H	Get Bus Architecture Function
61H	Return All POS IDs
62H	Return All EISA IDs

TESTCFG_SYS_GETBIOSADAPTER (40h) - Get Copy of BIOS/Ad

TESTCFG_SYS_GETBIOSADAPTER (40h) - Description

This function provides a copy of the contents of adapter memory.

TESTCFG_SYS_GETBIOSADAPTER (40h) - Command

Command
Must be zero.

TESTCFG_SYS_GETBIOSADAPTER (40h) - Addr0

Addr0
This field is interpreted as a physical address. It must be in a range of hex C0000 to hex FFFFF.

TESTCFG_SYS_GETBIOSADAPTER (40h) - NumBytes

NumBytes
Number of bytes of memory to copy.

TESTCFG_SYS_GETBIOSADAPTER (40h) - Parameter Packet Fo

Field	Length	C Datatype
Command	DWORD	ULONG
Addr0	DWORD	ULONG
NumBytes	WORD	USHORT

TESTCFG_SYS_GETBIOSADAPTER (40h) - Contents

Contents
Contents of memory at a specified location.

TESTCFG_SYS_GETBIOSADAPTER (40h) - Data Packet Format

TESTCFG provides a copy of the contents of adapter memory.

Field	Length	C Datatype
Contents	BYTES	BYTE[]

TESTCFG_SYS_GETBIOSADAPTER (40h) - Remarks

The contents of memory will not be copied past the physical address hex FFFFF.

TESTCFG_SYS_GETBIOSADAPTER (40h) -

Category:
IOCTL_TESTCFG_SYS (80h)
Function:
TESTCFG_SYS_GETBIOSADAPTER (40h)
Description:
Get Copy of BIOS/Adapter

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

TESTCFG_SYS_ISSUEINIOINSTR (41h) - Issue an In I/O Instructio

TESTCFG_SYS_ISSUEINIOINSTR (41h) - Description

This function issues the following *IN* I/O instruction.

TESTCFG_SYS_ISSUEINIOINSTR (41h) - io_address

io_address
I/O address

TESTCFG_SYS_ISSUEINIOINSTR (41h) - data_width

data_width
The integer equals the number of bytes in transfer value.
1 - In BYTE
2 - In WORD
4 - In DWORD

TESTCFG_SYS_ISSUEINIOINSTR (41h) - Parameter Packet Form

Field	Length	C Datatype
io_address	WORD	USHORT
data_width	WORD	USHORT

TESTCFG_SYS_ISSUEINIOINSTR (41h) - Value

Value
Value read.

TESTCFG_SYS_ISSUEINIOINSTR (41h) - Data Packet Format

Field	Length	C Datatype
Value	DWORD	ULONG

TESTCFG_SYS_ISSUEINIOINSTR (41h) - Remarks

Ports below address hex 100 are not accessible. ERROR_INVALID_PARAMETER is returned when such a port is referenced.

TESTCFG_SYS_ISSUEINIOINSTR (41h) -

Category:
80h
Function:
TESTCFG_SYS_ISSUEINIOINSTR (41h)
Description:
Issue an In I/O Instruction

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - Issue an Out I/O Instr

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - Description

This function issues the following *Out* I/O instruction.

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - io_address

io_address
I/O address

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - data width

data width

The integer equals the number of bytes in transfer value.
1 - Out BYTE
2 - Out WORD
4 - Out DWORD

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - Parameter Packet Fo

Field	Length	C Datatype
io_address	WORD	USHORT
data width	WORD	USHORT

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - Value

Value

Data value to write.

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - Data Packet Format

None.

Field	Length	C Datatype
Value	DWORD	ULONG

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) - Remarks

Ports below address hex 100 are not accessible. ERROR_INVALID_PARAMETER is returned when such a port is referenced.

TESTCFG_SYS_ISSUEOUTIOINSTR (42h) -

Category:
IOCTL_TESTCFG_SYS (80h)
Function:
TESTCFG_SYS_ISSUEOUTIOINSTR (42h)
Description:
Issue an Out I/O Instruction

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

TESTCFG_SYS_GETBUSARCH (60h) - Get Bus Architecture

TESTCFG_SYS_GETBUSARCH (60h) - Description

This function provides a query function for determining bus architecture.

TESTCFG_SYS_GETBUSARCH (60h) - Command

Command
Select query function:0=Query bus architecture.

TESTCFG_SYS_GETBUSARCH (60h) - Parameter Packet Format

Field	Length	C Datatype
Command	DWORD	ULONG

TESTCFG_SYS_GETBUSARCH (60h) - bus_arch

bus_arch

0=ISA
1=Micro Channel
2=EISA

TESTCFG_SYS_GETBUSARCH (60h) - Data Packet Format

On return, filled in as follows: code. Query function 0 (bus architecture).

Field	Length	C Datatype
bus_arch	DWORD	ULONG

TESTCFG_SYS_GETBUSARCH (60h) - Remarks

None.

TESTCFG_SYS_GETBUSARCH (60h) -

Category:
IOCTL_TESTCFG_SYS (80h)
Function:
TESTCFG_SYS_GETBUSARCH (60h)
Description:
Get Bus Architecture

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

TESTCFG_SYS_GETALLPOSIDS (61h) - Return All POS IDs

TESTCFG_SYS_GETALLPOSIDS (61h) - Description

This function provides a list of POS IDs in a Micro Channel bus system.

TESTCFG_SYS_GETALLPOSIDS (61h) - Command

Command
Must be zero.

TESTCFG_SYS_GETALLPOSIDS (61h) - Parameter Packet Forma

Field	Length	C Datatype
Command	DWORD	ULONG

TESTCFG_SYS_GETALLPOSIDS (61h) - Data Packet Format

The data packet is an array of ten POS IDs as follows:

Field	Length
POS ID for slot 0	WORD
POS ID for slot 1	WORD
.	.
.	.
POS ID for slot 8	WORD

TESTCFG_SYS_GETALLPOSIDS (61h) - Remarks

Returns all pos_id[n]=0 in an ISA or EISA configuration.

TESTCFG_SYS_GETALLPOSIDS (61h) -

Category:
IOCTL_TESTCFG_SYS (80h)
Function:
TESTCFG_SYS_GETALLPOSIDS (61h)
Description:
Return All POS IDs

Select an item:
Description

Parameter Packet Format
Data Packet Format
Remarks

TESTCFG_SYS_GETALLEISAIDS (62h) - Return All EISA IDs

TESTCFG_SYS_GETALLEISAIDS (62h) - Description

This function provides a list of EISA IDs in each of the sots of an EISA bus system.

TESTCFG_SYS_GETALLEISAIDS (62h) - Command

Command
Must be zero.

TESTCFG_SYS_GETALLEISAIDS (62h) - Parameter Packet Forma

Field	Length	C Datatype
Command	DWORD	ULONG

TESTCFG_SYS_GETALLEISAIDS (62h) - Data Packet Format

The Data Packet contains a table of 16 EISA IDs (each ID being a four-character string) as follows:

Field	Length
Product ID for slot 0	4 BYTES
Product ID for slot 1	4 BYTES
Product ID for slot 2	4 BYTES
.	.
.	.
.	.
Product ID for slot 15	4 BYTES

TESTCFG_SYS_GETALLEISAIDS (62h) - Remarks

Returns all product_id[n] =0 in an ISA or Micro Channel configuration.

TESTCFG_SYS_GETALLEISAIDS (62h) -

Category:
IOCTL_TESTCFG_SYS (80h)
Function:
TESTCFG_SYS_GETALLEISAIDS (62h)
Description:
Return All EISA IDs

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

Category 80h Resource Manager IOCtl Commands

RESOURCE.SYS provides two IOCtls that allow a ring 3 application to obtain a "snapshot" of the Resource Management data structures. Obtaining a snapshot of the Resource Management data structures consists of the following two steps:

1. A data structure representing a depth-first traversal of the Resource Manager node structure is obtained.
2. For each node traversed, the following information is provided:

 A Resource Manager handle to access the node.
 The depth of the node in the tree structure.
3. A copy of each Resource Manager node can be obtained by supplying the node handle returned in Step 1.

Function	Description
01h	Get Resource Manager Node Data
02h	Enumerate Resource Manager Nodes

Get Resource Manager Node Data - Function 01h

Category 80h, Function 01h - Description

Returns the contents of the Resource Manager node indicated by the handle provided.

Category 80h, Function 01h - RMHandle

RMHandle
This field must be initialized to the handle of the Resource Manager node to be interrogated.

Category 80h, Function 01h - Parameter Packet Format

Field	Length
RMHandle	ULONG

Category 80h, Function 01h - RMNodeSize

RMNodeSize
Size of the Resource Manager node information returned.

Category 80h, Function 01h - RMNode

RMNode
This field is set to a structure describing the Resource Manager node and its associated resources.

```
RM_NODE struct
    {
      ULONG             VersionInfo;
      ULONG             NodeType;
      RMHANDLE          DriverHandle;

      union
      {
        PADAPTERSTRUCT  pAdapterNode;
        PDEVICESTRUCT   pDeviceNode;
        PLDEVSTRUCT     pLDevNode;
        PSYSNAMESTRUCT  pSysNameNode;
        PDRIVERSTRUCT   pDriver;
      };
      PRESOURCELIST     pResourceList;
    }
```

VersionInfo (ULONG)
Version of the Resource Management driver.

NodeType (ULONG)
The type of node to which the handle provided refers.

pAdapterNode (PADAPTERSTRUCT)
This field contains a pointer to a structure that describes the Resource Manager node. This structure is a copy of the structure that was provided when the node was created. The pointer type selected from the union should be based on the NodeType value returned.

NodeType	Structure Pointer	Service
RMTYPE_ADAPTER	pAdapterNode	RMCreateAdapter
RMTYPE_DEVICE	pDeviceNode	RMCreateDevice
RMTYPE_LDEV	pLDevNode	RMCreateLDev
RMTYPE_SYSNAME	pSysNameNode	RMCreateSysName
RMTYPE_DRIVER	pDriver	RMCreateDriver

pResourceList (PRESOURCELIST)
Pointer to a structure containing a count and list of resource assigned to this node.

```
PRESOURCELIST struct
    {
      ULONG          Count;
      RESOURCESTRUCT Resource[1];
    }
```

Count (ULONG)
Count of resource structures returned.

Resource[] (RESOURCESTRUCT)
An array of resource structures assigned to this node. See RESOURCELIST for a description of the RESOURCESTRUCT data type.

Category 80h, Function 01h - Data Packet Format

All data packet fields are output fields for this function.

Field	Length
RMNodeSize	ULONG
RMNode	

Category 80h, Function 01h - Remarks

None.

Category 80h, Function 01h -

Category:
80h
Function:
01h
Description:
Get Resource Manager Node Data

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

Enumerate Resource Manager Nodes - Function 02h

Category 80h, Function 02h - Description

Traverses the Resource Manager node structure and returns the results of the traversal as a list of Resource Manager handles and traversal depth.

Category 80h, Function 02h - Command

Command
Traverses the physical device tree. This traversal reports all adapters and devices registered with the Resource Manager.

Category 80h, Function 02h - Parameter Packet Format

This field must indicate the type of traversal being requested.

Field	Length
Command	WORD

Category 80h, Function 02h - NumEntries

NumEntries
This field reports the number of node entries traversed.

Category 80h, Function 02h - NodeEntry[]

NodeEntry[]
This field is an array of the following structure:

```
NODEENTRY struct
           {
            RMHANDLE RMHandle;
            ULONG    Depth;
           };
```

RMHandle (RMHANDLE)
Resource Manager handle of the node traversed.

Depth (ULONG)
Level of the tree structure on which the node resides.

Category 80h, Function 02h - Data Packet Format

All data packet fields are output fields for this function.

Field	Length
NumEntries	ULONG
NodeEntry[]	8 * NumEntries

Category 80h, Function 02h - Remarks

None.

Category 80h, Function 02h -

Category:
80h
Function:
02h
Description:
Enumerate Resource Manager Nodes

Select an item:
Description

Parameter Packet Format
Data Packet Format
Remarks

Category 80h CD-ROM Drive and Disc IOCtl Commands

The OS/2 CD-ROM Device Manager provides an interface through generic IOCtls.

The CD-ROM device driver returns error values in the range of hex FF00 through FF14. DOS DevIOCtl return codes are described in the OS/2 Programming Reference manuals.

Function	Description
40h	Reset Drive
44h	Eject Disc
45h	Close Tray
46h	Lock/Unlock Door
50h	Seek
60h	Return Device Status
61h	Identify CD-ROM Driver
63h	Return Sector Size
70h	Report Location of Drive Head
72h	Read Long
78h	Return Volume Size
79h	Get UPC

CDROMDISK_RESETDRIVE (40h) - Reset Drive

CDROMDISK_RESETDRIVE (40h) - Description

This function resets and reinitializes the drive.

CDROMDISK_RESETDRIVE (40h) - Signature

Signature

Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_RESETDRIVE (40h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_RESETDRIVE (40h) - Data Packet Format

None.

CDROMDISK_RESETDRIVE (40h) - Returns

13h	Unsupported parameter. The first 4 bytes of the parameter packet are not "Signature CD01".
14h	Device already in use.

CDROMDISK_RESETDRIVE (40h) - Remarks

None.

CDROMDISK_RESETDRIVE (40h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_RESETDRIVE (40h)
Description:
Reset Drive

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_EJECTDISK (44h) - Eject Disc

CDROMDISK_EJECTDISK (44h) - Description

This function opens the drive door and ejects the disc for removal. No error is returned if the door is already open.

CDROMDISK_EJECTDISK (44h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_EJECTDISK (44h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_EJECTDISK (44h) - Data Packet Format

None.

CDROMDISK_EJECTDISK (44h) - Returns

13h	Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".
14h	Device already in use.

CDROMDISK_EJECTDISK (44h) - Remarks

None.

CDROMDISK_EJECTDISK (44h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_EJECTDISK (44h)
Description:
Eject Disc

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_CLOSETRAY (45h) - Close Tray

CDROMDISK_CLOSETRAY (45h) - Description

Closes the drive tray for CD-ROM drives with tray-loading mechanisms. No error is returned if the tray is already closed.

CDROMDISK_CLOSETRAY (45h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_CLOSETRAY (45h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_CLOSETRAY (45h) - Data Packet Format

None.

CDROMDISK_CLOSETRAY (45h) - Returns

13h	Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".
14h	Device already in use.

CDROMDISK_CLOSETRAY (45h) - Remarks

None.

CDROMDISK_CLOSETRAY (45h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_CLOSETRAY (45h)
Description:
Close Tray

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_LOCKUNLOCKDOOR (46h) - Lock/Unlock Door

CDROMDISK_LOCKUNLOCKDOOR (46h) - Description

This function locks and unlocks the drive door. When the drive door is locked, it cannot be opened from the front control panel of the drive. This command can be issued while the drive is playing.

CDROMDISK_LOCKUNLOCKDOOR (46h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_LOCKUNLOCKDOOR (46h) - Lock/Unlock Flag

Lock/Unlock Flag
Indicates whether to Lock or Unlock.
00=Unlock
01=Lock

CDROMDISK_LOCKUNLOCKDOOR (46h) - Parameter Packet For

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]
Lock/Unlock Flag	BYTE	UCHAR

CDROMDISK_LOCKUNLOCKDOOR (46h) - Data Packet Format

None.

CDROMDISK_LOCKUNLOCKDOOR (46h) - Returns

13h	Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_LOCKUNLOCKDOOR (46h) - Remarks

None.

CDROMDISK_LOCKUNLOCKDOOR (46h) -

Category:
IOCTL_CDROMDISK (80h)

Function:
CDROMDISK_LOCKUNLOCKDOOR (46h)
Description:
Lock/Unlock Door

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_SEEK (50h) - Seek

CDROMDISK_SEEK (50h) - Description

This function moves the read head to the specified block. Subsequent requests to move the head must wait until the execution of this command is complete.

CDROMDISK_SEEK (50h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_SEEK (50h) - Addressing Mode

Addressing Mode
The addressing format of the Starting Sector Number.
00=Logical-block format
01=Minutes/seconds/frame format

CDROMDISK_SEEK (50h) - Starting Sector Number

Starting Sector Number
The Starting Sector Number to seek to.

CDROMDISK_SEEK (50h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]
Addressing Mode	BYTE	UCHAR
Starting Sector Number	DWORD	ULONG

CDROMDISK_SEEK (50h) - Data Packet Format

None.

CDROMDISK_SEEK (50h) - Returns

02H	Device not ready: The drive is not ready, or the controller does not respond.
06H	Seek error: The drive does not seek.
07H	Unknown media: The media is a compact disc that is not formatted according to the CD Redbook specifications.
08H	Sector not found: The requested sector is not on the disc. This error is returned if a data address cannot be found or if the minutes/seconds/frame address cannot be found.
10H	Uncertain media.
13H	Unsupported parameter: The addressing mode is out of range, or the first 4 bytes of the parameter packet are not "Signature CD01".
14H	Device already is use.

CDROMDISK_SEEK (50h) - Remarks

None.

CDROMDISK_SEEK (50h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_SEEK (50h)
Description:
Seek

Select an item:
Description

Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_DEVICESTATUS (60h) - Device Status

CDROMDISK_DEVICESTATUS (60h) - Description

This function returns the drive-and-door-open status as a doubleword set of bits in the data buffer. This function is performed regardless of whether the drive is already playing.

CDROMDISK_DEVICESTATUS (60h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_DEVICESTATUS (60h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_DEVICESTATUS (60h) - Data Packet Format

```
            Dword (ULONG) Returned status bits

31          Reserved; always set to 0

30          0=Does not support reading of CD-DA sectors
            1=Supports reading of CD-DA sectors

13-29       Reserved; always set to 0

12          0=Drive is not playing audio
            1=Drive is playing back audio information; while
            in the playback mode, data-related commands will
```

```
            return a device-already-in-use indicator.

 11         0=Disc is present
            1=Disc is not present

 10         0=Does not support reading of Mode 2 sectors
            1=Supports reading of Mode 2 sectors
            (Form 1 and Form 2)

 9          0=Supports logical-block addressing
            1=Supports logical-block and minute-
            second-frame addressing

 8          0=No audio-channel manipulation
            1=Supports audio-channel manipulation

 7          0=Prefetching not supporting
            1=Prefetchng supported internally

 6          Reserved; always set to 0

 5          0=No interleaving support
            1=ISO-9660 interleaving size and skip supported

 4          0=Drive can perform only data reads
            1=Drive can read data and play audio,
            video, or both

 3          0=Read only
            1=Read and write operations supported
            by drive

 2          0=Supports only 2048-byte sectors (cooked)
            1=Supports 2048-byte and 2352-byte sectors
              (cooked and raw)

 1          0=Door locked
            1=Door unlocked

 0          0=Door closed
            1=Door open
```

CDROMDISK_DEVICESTATUS (60h) - Returns

02h	Device not ready: The drive is not ready, or the controller does not respond.
13h	Unsupported parameter: The addressing mode is out of range, or the first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_DEVICESTATUS (60h) - Remarks

None.

CDROMDISK_DEVICESTATUS (60h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_DEVICESTATUS (60h)

Description:
Device Status

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_GETDRIVER (61h) - Identify CD-ROM Driver

CDROMDISK_GETDRIVER (61h) - Description

This function identifies the specified device driver as a valid CD-ROM driver. File system drivers (FSDs) and applications should call this routine before issuing IOCtls that could lead to confusion or data damage if received by a non-CD-ROM driver. This function is performed regardless of whether the drive is already playing.

Note: Any two valid ASCII digits, each in the range of hex 30 to 39, can be used after the "CD" to indicate the caller wanting to identify the CD-ROM driver.

CDROMDISK_GETDRIVER (61h) - Signature

Signature
Signature string of "CDNN". (Each N must be an ASCII digit 0 - 9).

CDROMDISK_GETDRIVER (61h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_GETDRIVER (61h) - Signature

Signature
Return signature string of "CD01".

CDROMDISK_GETDRIVER (61h) - Data Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_GETDRIVER (61h) - Returns

13H Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_GETDRIVER (61h) - Remarks

None.

CDROMDISK_GETDRIVER (61h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_GETDRIVER (61h)
Description:
Identify CD-ROM Driver

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_GETSECTORSIZE (63h) - Return Sector Size

CDROMDISK_GETSECTORSIZE (63h) - Description

This function returns in the data buffer the number of bytes per sector that the device driver supports. This function is performed regardless of whether the drive is already playing.

CDROMDISK_GETSECTORSIZE (63h) - Signature

Signature

Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_GETSECTORSIZE (63h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_GETSECTORSIZE (63h) - Sector Size

Sector Size
The number of bytes per sector used in read operations.

CDROMDISK_GETSECTORSIZE (63h) - Data Packet Format

Field	Length	C Datatype
Sector Size	WORD	USHORT

CDROMDISK_GETSECTORSIZE (63h) - Returns

13h	Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_GETSECTORSIZE (63h) - Remarks

None.

CDROMDISK_GETSECTORSIZE (63h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_GETSECTORSIZE (63h)
Description:
Return Sector Size

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_GETHEADLOC (70h) - Report Location of Drive Hea

CDROMDISK_GETHEADLOC (70h) - Description

Reports the current drive-head location in the specified addressing mode. This function is performed regardless of whether the drive is already playing. Programs that play audio can make repeated calls to this function to display the current play position.

CDROMDISK_GETHEADLOC (70h) - Signature

Signature

Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_GETHEADLOC (70h) - Addressing Mode

Addressing Mode
Address format returned in Head Location.
00=Logical Block Format

01=Minutes/Seconds/Frame Format

CDROMDISK_GETHEADLOC (70h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]
Addressing Mode	BYTE	UCHAR

CDROMDISK_GETHEADLOC (70h) - Head Location

Head Location
Current drive-head Location.

CDROMDISK_GETHEADLOC (70h) - Data Packet Format

Field	Length	C Datatype
Head Location	DWORD	ULONG

CDROMDISK_GETHEADLOC (70h) - Returns

The error return codes for this function are as follows:

02h	Device not ready: The drive is not ready, or the controller does not respond.
07h	Unknown Media: The media is a compact disc that is not formatted according to the CD Redbook specification.
10h	Uncertain media.
13h	Unsupported parameter: The addressing mode is out of range or not supported, or the first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_GETHEADLOC (70h) - Remarks

None.

CDROMDISK_GETHEADLOC (70h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_GETHEADLOC (70h)
Description:
Report Location of Drive Head

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_READLONG (72h) - Read Long

CDROMDISK_READLONG (72h) - Description

This function reads 2352 bytes of the specified sectors in the data buffer. The returned sectors include the 12 bytes of sync, 4 bytes of header, 2048 bytes of data, and 288 bytes of EDC/ECC at the end.

CDROMDISK_READLONG (72h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_READLONG (72h) - Addressing Mode

Addressing Mode
Addressing format of Starting Sector Number
00=Logical Block Format
01=Minutes/Seconds/Frame Format

CDROMDISK_READLONG (72h) - Number of Sectors

Number of Sectors
The number of 2352-byte sectors to read. Must not be 0.

CDROMDISK_READLONG (72h) - Starting Sector Number

Starting Sector Number
The starting sector number of the read operation.

CDROMDISK_READLONG (72h) - Reserved

Reserved
Must be 0.

CDROMDISK_READLONG (72h) - Interleave size

Interleave size
Not used. Must be 0.

CDROMDISK_READLONG (72h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]
Addressing Mode	BYTES	UCHAR[]
Number of Sectors	WORD	USHORT
Starting Sector Number	BYTE	UCHAR
Reserved	BYTE	UCHAR
Interleave size	BYTE	UCHAR

CDROMDISK_READLONG (72h) - Data Packet Format

Field	Length	C Datatype
Sync	12 BYTES	UCHAR[12]
Header	4 BYTES	UCHAR[4]
Data Area	2048 BYTES	UCHAR[2048]
EDC/ECC	288 BYTES	UCHAR[288]

CDROMDISK_READLONG (72h) - Returns

The error return codes for this function are as follows:

02h	Device not ready: The drive is not ready, or the controller does not respond.
03h	Unknown command: The drive does not support this request.
08h	Sector not found: The requested sector is not on the disc. This error is returned if a data address cannot be found or if the minutes/seconds/frame address cannot be found.
10h	Uncertain media.
13h	Unsupported parameter: The addressing mode is out of range or not supported, or if the first 4 bytes of the parameter buffer are not "Signature".
14h	Device already in use.

CDROMDISK_READLONG (72h) - Remarks

None.

CDROMDISK_READLONG (72h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_READLONG (72h)
Description:
Read Long

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_GETVOLUMESIZE (78h) - Return Volume Size

CDROMDISK_GETVOLUMESIZE (78h) - Description

This function returns the total number of accessible sectors on the disc. The value returned is the same as that returned in the extended BIOS parameter block by DSK_GETDEVICEPARAMS. This function is performed regardless of whether the drive is already playing.

CDROMDISK_GETVOLUMESIZE (78h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_GETVOLUMESIZE (78h) - Parameter Packet Forma

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_GETVOLUMESIZE (78h) - Volume Size

Volume Size
The size of the volume in sectors.

CDROMDISK_GETVOLUMESIZE (78h) - Data Packet Format

Field	Length	C Datatype
Volume Size	DWORD	ULONG

CDROMDISK_GETVOLUMESIZE (78h) - Returns

The error return codes for this function are as follows:

02h	Device not ready: The drive is not ready, or the controller does not respond.
07h	Unknown media: The media is a compact disc that is not formatted according to CD Redbook specifications.
10h	Uncertain media.
13h	Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_GETVOLUMESIZE (78h) - Remarks

None.

CDROMDISK_GETVOLUMESIZE (78h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_GETVOLUMESIZE (78h)
Description:
Return Volume Size

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMDISK_GETUPC (79h) - Get UPC

CDROMDISK_GETUPC (79h) - Description

This function returns the Universal Product Code (UPC) for the disc. This is a unique code identifying the disc. The UPC is 13 successive BCD digits (4 bits each) followed by 12 bits set to 0.

CDROMDISK_GETUPC (79h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMDISK_GETUPC (79h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMDISK_GETUPC (79h) - Control ADR

Control ADR

The Track-Control Information byte corresponds to the byte in the TOC lead-in track containing the two 4-bit control field and ADR. The Control information is in the most-significant 4 bits, and the ADR information is in the least-significant 4 bits.

CDROMDISK_GETUPC (79h) - Universal Product Code

Universal Product Code

Universal Product Code. If no UPC was encoded on the disc, then zeroes are returned to this field.

CDROMDISK_GETUPC (79h) - Reserved

Reserved

Must be 0.

CDROMDISK_GETUPC (79h) - Frame

Frame

Frame number of current head location.

CDROMDISK_GETUPC (79h) - Data Packet Format

Field	Length	C Datatype
Control ADR	BYTE	UCHAR
Universal Product Code	7 BYTES	UCHAR[7]
Reserved	BYTE	UCHAR
Frame	BYTE	UCHAR

CDROMDISK_GETUPC (79h) - Returns

The error return codes for this function are as follows:

02h	Device not ready: The drive is not ready, or the controller does not respond.
03h	Unknown command: The drive does not support this request.
13h	Unsupported parameter: The first 4 bytes of the parameter packet are not "Signature".

CDROMDISK_GETUPC (79h) - Remarks

None.

CDROMDISK_GETUPC (79h) -

Category:
IOCTL_CDROMDISK (80h)
Function:
CDROMDISK_GETUPC (79h)
Description:
Get UPC

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 80h High-Resolution Timer IOCtl Commands

The following IOCtls are defined and supported by the TIMER0$ device driver by way of the DosDevIOCtl call. The IOCtl category code is 80h (defined as HRT_IOCTL_CATEGORY).

The function codes within the HRT_IOCTL_CATEGORY are:

Function	Description

```
00h        Get Version

01h        Get Resolution

02h        Set Resolution

03h        Get Pointer to Clock Timer

04h        Free Pointer to Clock Timer

05h        Block Until Time Elapses

06h-7Fh    Reserved
```

An example of the DosDevIOCtl calling convention for reading the current time follows:

```
#include "tmr0_ioc.h"

APIRET rc           = NO_ERROR;
HFILE  hfile        = NULLHANDLE;
ULONG  ulAction     = 0L;

ULONG  ulResolution = 1;
ULONG  ulSize1      = sizeof(ulResolution);

ULONG  * _Seg16 pulTimer16;             // defines a 16:16 pointer
ULONG  ulSize2      = sizeof(pulTimer16);
ULONG  *pulTimer;

rc=DosOpen( "TIMER0$ ",
            &hfile,
            &ulAction,
            0,0,
            OPEN_ACTION_OPEN_IF_EXISTS,
            OPEN_FLAGS_FAIL_ON_ERROR | OPEN_SHARE_DENYNONE |
            OPEN_ACCESS_READWRITE,
            NULL);
if (rc) {
   printf("Couldn't open TIMER0$.\n");
   return;
}

printf("TIMER0$ opened. File Handle is %lu\n",hfile);

rc=DosDevIOCtl( hfile,
                HRT_IOCTL_CATEGORY,
                HRT_SET_RESOLUTION,
                &ulResolution,
                ulSize1,
                &ulSize1,
                NULL,
                0,
                NULL);
if (rc) {
   printf("Couldn't set resolution.\n");
   DosClose(hfile);
   return;
}

rc=DosDevIOCtl( hfile,
                HRT_IOCTL_CATEGORY,
                HRT_GET_POINTER,
                NULL,
                0,
                NULL,
                &pulTimer16,
                ulSize2,
                &ulSize2);
if (rc) {
   printf("Couldn't get pointer.\n");
   DosClose(hfile);
   return;
}

pulTimer = pulTimer16;    // converts a 16:16 pointer to a 0:32 pointer
if (!pulTimer) {
   printf("NULL pointer.\n");
```

```
   DosClose(hfile);
   return;
}

// At this point, pulTimer is now a pointer to the timer 0 counter variable.

rc=DosClose(hfile);
```

The DosOpen of TIMER0$ registers the application as a client. At this point, the high-resolution timer (HRT) is taking interrupts, so only open the driver when timer services are needed. The pointer is valid for the life of the process, and each call to HRT_GET_POINTER allocates another selector, so this call should only be made once.

While the HRT is taking interrupts, DOS sessions (VDMs) will not receive INT 8 or INT 1Ch calls. The reason for this is that OS/2 only allows one device driver to receive interrupts on a given IRQ, and VTIMER.SYS (the VDD that provides INT 8 and INT 1Ch services to VDM's) expects CLOCK0x.SYS to deliver IRQ 0 interrupts. When the HRT is active, it receives the IRQ 0 interrupts instead of CLOCK0x.SYS, and thus VTIMER.SYS never gets them either.

Note: Performance tools, such as C Set's DDE4XTRA.SYS and VisualAge's CPPOPA3.SYS, are incompatible with this driver. Performance analysis tools cannot be run while the high-resolution timer is active, and vice versa.

The following code segment illustrates how to block a time-critical thread until a certain time has eleapsed. Return code checking has been omitted for brevity. A time-critical thread should be used for this function.

```
ULONG ulDelay = 1;               // Number of milliseconds to wait
ULONG ulSize2 = sizeof(ulDelay);

DosOpen("TIMER0$ ", &hfile,
        &ulAction,0,0,ulOpenFlag,ulOpenMode,NULL);

DosSetPriority(0,PRTYC_TIMECRITICAL,0,0);

DosDevIOCtl( hfile,
             HRT_IOCTL_CATEGORY,
             HRT_BLOCKUNTIL,
             &ulDelay,
             ulSize2,
             &ulSize2,
             NULL,
             0,
             NULL);
```

HRT_QUERY_VERSION (00h) - Query Version

HRT_QUERY_VERSION (00h) - Description

This function returns the version of the high resolution timer support that is installed.

HRT_QUERY_VERSION (00h) - Parameter Packet Format

None.

HRT_QUERY_VERSION (00h) - Version

Version

High-resolution timer device driver version.

The 16 bits are layed out in this format:

```
  2 bits    4 bits    4 bits    2 bits    4 bits


  Major     Minor      Bug      Phase     Build
                      Fix #               Level
```

Field Description	Valid Values
Major Version Number	1, 2, 3, or 4
Minor Version Number	0 to 15
Bug Fix Number	0 to 15
Development Phase	00h - HRTVER_DEVELOP 01h - HRTVER_ALPHA 02h - HRTVER_BETA
Build Level	0 to 15

The macros HRTVER_MAJOR(), HRTVER_MINOR(), HRTVER_BUGFIX(), HRTVER_PHASE(), and HRTVER_BUILD() can be used to extract the various pieces from the 16-bit device driver version information.

HRT_QUERY_VERSION (00h) - Data Packet Format

Field	Length	C Datatype
Version	WORD	USHORT

HRT_QUERY_VERSION (00h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	HRTERR_BADPARAM

HRT_QUERY_VERSION (00h) - Remarks

This function returns the version of the high-resolution timer driver.

HRT_QUERY_VERSION (00h) -

Category:
HRT_IOCTL_CATEGORY (80h)
Function:
HRT_QUERY_VERSION (00h)
Description:
Query Version

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

HRT_GET_RESOLUTION (01h) - Get Resolution

HRT_GET_RESOLUTION (01h) - Description

This function returns the resolution of the high resolution timer.

HRT_GET_RESOLUTION (01h) - Parameter Packet Format

None.

HRT_GET_RESOLUTION (01h) - Resolution

Resolution
Resolution of high-resolution timer in milliseconds.

HRT_GET_RESOLUTION (01h) - Data Packet Format

Field	Length	C Datatype
Resolution	DWORD	ULONG

HRT_GET_RESOLUTION (01h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	HRTERR_BADPARAM

HRT_GET_RESOLUTION (01h) - Remarks

This function returns the resolution of the high-resolution timer in milliseconds.

HRT_GET_RESOLUTION (01h) -

Category:
HRT_IOCTL_CATEGORY (80h)
Function:
HRT_GET_RESOLUTION (01h)
Description:
Get Resolution

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

HRT_SET_RESOLUTION (02h) - Set Resolution

HRT_SET_RESOLUTION (02h) - Description

Sets the resolution for the high-resolution timer driver.

HRT_SET_RESOLUTION (02h) - Resolution

Resolution
The resolution requested for the high-resolution timer, in milliseconds.

HRT_SET_RESOLUTION (02h) - Parameter Packet Format

This function sets the resolution of the high-resolution timer.

Field	Length
Resolution	DWORD

HRT_SET_RESOLUTION (02h) - Data Packet Format

None

HRT_SET_RESOLUTION (02h) - Remarks

The application should use this IOCtl to tell the high-resolution timer driver what resolution it needs. By selecting a value higher than 1 ms, the driver may be able to conserve system resources while still providing accurate enough timing.

HRT_SET_RESOLUTION (02h) -

Category:
HRT_IOCTL_CATEGORY (80h)
Function:
HRT_SET_RESOLUTION (02h)
Description:
Set Resolution

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

HRT_GET_POINTER (03h) - Get Pointer

HRT_GET_POINTER (03h) - Description

This function returns a pointer to the clock counter.

HRT_GET_POINTER (03h) - Parameter Packet Format

None.

HRT_GET_POINTER (03h) - Pointer to Clock Timer

Pointer to Clock Timer
Pointer (16:16 format) to the current clock timer. Datatype for this pointer is ULONG * _Seg16.

HRT_GET_POINTER (03h) - Data Packet Format

Field	Length	C Datatype
Pointer to Clock Timer	DWORD	ULONG * _Seg16

HRT_GET_POINTER (03h) - Returns

Possible values are shown in the following list:

0	NO_ERROR
1	HRTERR_BADPARAM

HRT_GET_POINTER (03h) - Remarks

This function returns a pointer to the current clock timer. This pointer is valid for the life of the process or until the pointer is freed by using HRT_FREE_POINTER, or the high-resolution timer driver is closed by DosClose. The pointer is freed at process termination or when DosClose is issued.

HRT_GET_POINTER (03h) -

Category:
HRT_IOCTL_CATEGORY (80h)
Function:
HRT_GET_POINTER (03h)
Description:
Get Pointer

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

HRT_FREE_POINTER (04h) - Free Pointer

HRT_FREE_POINTER (04h) - Description

Frees the pointer to the clock timer previously obtained using HRT_GET_POINTER.

HRT_FREE_POINTER (04h) - Pointer to clock timer

Pointer to clock timer
The pointer to the clock timer previously obtained.

HRT_FREE_POINTER (04h) - Parameter Packet Format

This function frees the pointer to the clock timer.

Field	Length
Pointer to clock timer	DWORD

HRT_FREE_POINTER (04h) - Data Packet Format

None

HRT_FREE_POINTER (04h) - Remarks

The pointer is valid for the life of the process, or until the pointer is freed with this call.

HRT_FREE_POINTER (04h) -

Category:
HRT_IOCTL_CATEGORY (80h)
Function:
HRT_FREE_POINTER (04h)
Description:
Free Pointer

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

HRT_BLOCKUNTIL (05h) - Block Until Time Elapses

HRT_BLOCKUNTIL (05h) - Description

Causes the current thread to block until the specified time elapses.

HRT_BLOCKUNTIL (05h) - Time to block

Time to block
The amount of time, in milliseconds, to block.

HRT_BLOCKUNTIL (05h) - Parameter Packet Format

This function causes the current thread to block until the specified amount of time elapses.

Field	Length
Time to block	DWORD

HRT_BLOCKUNTIL (05h) - Data Packet Format

None

HRT_BLOCKUNTIL (05h) - Remarks

A time-critical thread should be used for this call.

```
ULONG ulDelay = 1;               // Number of milliseconds to wait
ULONG ulSize2 = sizeof(ulDelay);

DosOpen("TIMER0$ ", &hfile,
        &ulAction,0,0,ulOpenFlag,ulOpenMode,NULL);

DosSetPriority(0,PRTYC_TIMECRITICAL,0,0);

DosDevIOCtl( hfile,
             HRT_IOCTL_CATEGORY,
             HRT_BLOCKUNTIL,
             &ulDelay,
             ulSize2,
             &ulSize2,
             NULL,
             0,
             NULL);
```

HRT_BLOCKUNTIL (05h) -

Category:
HRT_IOCTL_CATEGORY (80h)
Function:
HRT_BLOCKUNTIL (05h)
Description:
Block Until Time Elapses

Select an item:
Description
Parameter Packet Format
Data Packet Format
Remarks

Category 81h CD-ROM Audio IOCtl Commands

The OS/2 CD-ROM Device Manager provides an interface through generic IOCtls.

The CD-ROM device driver returns error values in the range of hex FF00 through FF14. DOSDevIOCtl return codes are described in the OS/2 Programming Reference manuals.

Function	Description
40h	Set Audio Channel Control
50h	Play Audio
51h	Stop Audio
52h	Resume Audio
60h	Return Audio-Channel Information
61h	Return Audio-Disk Information

62h	Return Audio-Track Information
63h	Return Audio-Subchannel Q Information
65h	Return Audio-Status Information

CDROMAUDIO_SETCHANNELCTRL (40h) - Set Audio Channel Co

CDROMAUDIO_SETCHANNELCTRL (40h) - Description

This function sets the playback control of the audio channels. This includes assigning input channels to output channels and controlling the volume of each output channel. This function is performed regardless of whether the drive is already playing.

CDROMAUDIO_SETCHANNELCTRL (40h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_SETCHANNELCTRL (40h) - Parameter Packet For

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_SETCHANNELCTRL (40h) - Data Packet Format

Field	Length	C Datatype
Input channel for output channel 0	BYTE	UCHAR
Volume control (0-FFh) for output channel 0	BYTE	UCHAR
Input channel for output	BYTE	UCHAR

channel 1		
Volume control (0-FFh) for output channel 1	BYTE	UCHAR
Input channel for output channel 2	BYTE	UCHAR
Volume control (0-FFh) for output channel 2	BYTE	UCHAR
Input channel for output channel 3	BYTE	UCHAR
Volume control (0-FFh) for output channel 3	BYTE	UCHAR

CDROMAUDIO_SETCHANNELCTRL (40h) - Returns

02H	Device not ready: The drive is not ready, or the controller does not respond.
07H	Unknown media: The media is a compact disc that is not formatted according to the CD Redbook specifications.
13H	Unsupported parameter: Any of the input channels are out of range, or the first 4 bytes of the parameter buffer are not "CD01."

CDROMAUDIO_SETCHANNELCTRL (40h) - Remarks

None.

CDROMAUDIO_SETCHANNELCTRL (40h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_SETCHANNELCTRL (40h)
Description:
Set Audio Channel Control

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_PLAYAUDIO (50h) - Play Audio

CDROMAUDIO_PLAYAUDIO (50h) - Description

This function plays the selected audio tracks until the selection is played or until a Stop Audio command (Category 81h, Function 51h) is received. If the ending sector number is reached, or if the end of the disc or the end of the audio section is reached, the drive will stop playing. This is not considered an error.

CDROMAUDIO_PLAYAUDIO (50h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_PLAYAUDIO (50h) - Addressing Mode

Addressing Mode
Addressing format used for Starting Sector Number and Ending Sector Number.
00=Logical block format
01=Minutes/seconds/frame format

CDROMAUDIO_PLAYAUDIO (50h) - Starting Sector Number

Starting Sector Number
The starting sector of the play operation.

CDROMAUDIO_PLAYAUDIO (50h) - Ending Sector Number

Ending Sector Number
The ending sector of the play operation.

CDROMAUDIO_PLAYAUDIO (50h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

Addressing Mode	BYTE	UCHAR
Starting Sector Number	DWORD	ULONG
Ending Sector Number	DWORD	ULONG

CDROMAUDIO_PLAYAUDIO (50h) - Data Packet Format

None.

CDROMAUDIO_PLAYAUDIO (50h) - Returns

02H	Device not ready. The drive is not ready, or the controller does not respond.
07H	Unknown media. The media is a compact disc that is not formatted according to the CD Redbook specifications.
08H	Sector not found. The requested sector is not on the disc. This error is returned if a data address cannot be found or if the minutes/seconds/frame address cannot be found.
10H	Uncertain media.
13H	Unsupported parameter. The first 4 bytes of the parameter buffer are not "CD01" or the addressing mode is out of range.
14H	Device already in use.

CDROMAUDIO_PLAYAUDIO (50h) - Remarks

None.

CDROMAUDIO_PLAYAUDIO (50h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_PLAYAUDIO (50h)
Description:
Play Audio

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_STOPAUDIO (51h) - Stop Audio

CDROMAUDIO_STOPAUDIO (51h) - Description

This function cancels any active play request. If the drive is not playing, this command returns without an error.

CDROMAUDIO_STOPAUDIO (51h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_STOPAUDIO (51h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_STOPAUDIO (51h) - Data Packet Format

None.

CDROMAUDIO_STOPAUDIO (51h) - Returns

13h	Unsupported parameter. The first 4 bytes of the parameter buffer are not "CD01" or the addressing mode is out of range.

CDROMAUDIO_STOPAUDIO (51h) - Remarks

None.

CDROMAUDIO_STOPAUDIO (51h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_STOPAUDIO (51h)
Description:
Stop Audio

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_RESUMEAUDIO (52h) - Resume Audio

CDROMAUDIO_RESUMEAUDIO (52h) - Description

This function resumes playing audio tracks after play has been interrupted by the Stop Audio command. An error is returned if there was no previous Stop Audio command (Category 81h, Function 51h).

CDROMAUDIO_RESUMEAUDIO (52h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_RESUMEAUDIO (52h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_RESUMEAUDIO (52h) - Data Packet Format

None.

CDROMAUDIO_RESUMEAUDIO (52h) - Returns

02h	Device not ready. The drive is not ready, or the controller does not respond.
07h	Unknown media. The media is a compact disc that is not formatted according to the CD Redbook specifications.
08h	Sector not found. There was no previous Stop Audio command.
10h	Uncertain media.
13h	Unsupported parameter. The first 4 bytes of the parameter buffer are not "CD01".

CDROMAUDIO_RESUMEAUDIO (52h) - Remarks

None.

CDROMAUDIO_RESUMEAUDIO (52h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_RESUMEAUDIO (52h)
Description:
Resume Audio

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_GETCHANNEL (60h) - Return Audio-Channel Infor

CDROMAUDIO_GETCHANNEL (60h) - Description

This function returns the current settings of the audio-channel controls. These are either the default settings or those set with the Audio-Channel Control function (Category 81h, Function 40h). This command is performed regardless of whether the drive is already playing.

CDROMAUDIO_GETCHANNEL (60h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_GETCHANNEL (60h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_GETCHANNEL (60h) - Data Packet Format

Field	Length	C Datatype
Input channel for output channel 0	BYTE	UCHAR
Volume control (0-FFh) for output channel 0	BYTE	UCHAR
Input channel for output channel 1	BYTE	UCHAR
Volume control (0-FFh) for output channel 1	BYTE	UCHAR
Input channel for output channel 2	BYTE	UCHAR
Volume control (0-FFh) for output channel 2	BYTE	UCHAR
Input channel for output channel 3	BYTE	UCHAR
Volume control (0-FFh) for output channel 3	BYTE	UCHAR

CDROMAUDIO_GETCHANNEL (60h) - Returns

13H	Unsupported parameter. Any of the input channels are out of range, or the first 4 bytes of the parameter buffer are not "CD01".

CDROMAUDIO_GETCHANNEL (60h) - Remarks

None.

CDROMAUDIO_GETCHANNEL (60h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_GETCHANNEL (60h)
Description:
Return Audio-Channel Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_GETAUDIODISK (61h) - Return Audio-Disc Informa

CDROMAUDIO_GETAUDIODISK (61h) - Description

This function returns the first and last track numbers and the CD Redbook address for the lead-out track. This information is read from the TOC information on the subchannel Q lead-in track. The first and last track numbers are binary values, not BCD. This command is performed regardless of whether the play audio is processing.

CDROMAUDIO_GETAUDIODISK (61h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_GETAUDIODISK (61h) - Parameter Packet Format

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_GETAUDIODISK (61h) - First Track Number

First Track Number
First track number on the disc (in binary)

CDROMAUDIO_GETAUDIODISK (61h) - Last Track Number

Last Track Number
Last track number on the disc (in binary)

CDROMAUDIO_GETAUDIODISK (61h) - Lead-out Address

Lead-out Address
The address of the lead-out track in MSF format

CDROMAUDIO_GETAUDIODISK (61h) - Data Packet Format

Field	Length	C Datatype
First Track Number	BYTE	UCHAR
Last Track Number	BYTE	UCHAR
Lead-out Address	DWORD	ULONG

CDROMAUDIO_GETAUDIODISK (61h) - Returns

02H	Device not ready. The drive is not ready, or the controller does not respond.
07H	Unknown media. The media is a compact disc that is not formatted according to the CD Redbook specification.
10H	Uncertain media.
13H	Unsupported parameter. The first 4 bytes of the parameter buffer are not "CD01".

CDROMAUDIO_GETAUDIODISK (61h) - Remarks

None.

CDROMAUDIO_GETAUDIODISK (61h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_GETAUDIODISK (61h)
Description:
Return Audio-Disc Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_GETAUDIOTRACK (62h) - Return Audio-Track Info

CDROMAUDIO_GETAUDIOTRACK (62h) - Description

This function returns the CD-ROM Redbook address for the starting point of the track, and the track-control information for that track.

By finding the range of tracks through Audio-Disc Information (Category 81h, Function 61h), this function can be called for each track to determine its type and length.

CDROMAUDIO_GETAUDIOTRACK (62h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_GETAUDIOTRACK (62h) - Track Number

Track Number
Track Number to return track information.

CDROMAUDIO_GETAUDIOTRACK (62h) - Parameter Packet Form

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]
Track Number	BYTE	UCHAR

CDROMAUDIO_GETAUDIOTRACK (62h) - Track Address

Track Address

The starting address of the track in MSF format.

CDROMAUDIO_GETAUDIOTRACK (62h) - Track-Control Informati

Track-Control Information

The Track-Control Information byte corresponds to the byte in the TOC lead-in track containing the two 4-bit control field and ADR. The Control information is in the most-significant 4 bits, and the ADR information is in the least-significant 4 bits. The track-control information is encoded as shown in the following table:

Bit	Description
7	0 = Two-channel audio 1 = Four-channel audio
6	0 = Audio track 1 = Data track
5	0 = Digital copy prohibited 1 = Digital copy permitted
4	0 = Audio without preemphasis 1 = Audio with preemphasis
3-0	ADR data for track

CDROMAUDIO_GETAUDIOTRACK (62h) - Data Packet Format

Field	Length	C Datatype
Track Address	DWORD	ULONG

Track-Control Information	BYTE	UCHAR

CDROMAUDIO_GETAUDIOTRACK (62h) - Returns

02H	Device not ready. The drive is not ready, or the controller does not respond.
07H	Unknown media. The media is a compact disc that is not formatted according to the CD Redbook specification.
08H	Sector not found: The requested track is outside the range of tracks in the TOC.
10h	Uncertain media.
13h	Unsupported parameter. The first 4 bytes of the parameter packet are not "CD01" or the specified track number is out of range.

CDROMAUDIO_GETAUDIOTRACK (62h) - Remarks

None.

CDROMAUDIO_GETAUDIOTRACK (62h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_GETAUDIOTRACK (62h)
Description:
Return Audio-Track Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_GETSUBCHANNELQ (63h) - Return Audio-Subcha

CDROMAUDIO_GETSUBCHANNELQ (63h) - Description

This function reads and returns the address information from a channel of the current head location. The function does not interrupt the present state of the drive. One of its intended purposes is to monitor the information of the read head while the drive is playing audio tracks.

Note: The Running Time within a track and Running Time on a disc are returned in binary.

CDROMAUDIO_GETSUBCHANNELQ (63h) - Signature

Signature

Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_GETSUBCHANNELQ (63h) - Parameter Packet Fo

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_GETSUBCHANNELQ (63h) - Data Packet Format

Field	Length	C Datatype
Control and ADR byte	BYTE	UCHAR
Track Number (in BCD)	BYTE	UCHAR
Index (in BCD)	BYTE	UCHAR
Running time within a track (minutes)	BYTE	UCHAR
Running time within a track (seconds)	BYTE	UCHAR
Running time within a track (frame)	BYTE	UCHAR
00	BYTE	UCHAR
Running time on a disc (minutes)	BYTE	UCHAR
Running time on a disc (seconds)	BYTE	UCHAR
Running time on a disc (frame)	BYTE	UCHAR

CDROMAUDIO_GETSUBCHANNELQ (63h) - Returns

02H	Device not ready. The drive is not ready, or the controller does not respond.
07H	Unknown media. The media is a compact disc that is not formatted according to CD Redbook specification.
10H	Uncertain media.
13H	Unsupported parameter. The first 4 bytes of the parameter buffer are not "CD01".

CDROMAUDIO_GETSUBCHANNELQ (63h) - Remarks

None.

CDROMAUDIO_GETSUBCHANNELQ (63h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_GETSUBCHANNELQ (63h)
Description:
Return Audio-Subchannel Q Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

CDROMAUDIO_GETAUDIOSTATUS (65h) - Return Audio-Status I

CDROMAUDIO_GETAUDIOSTATUS (65h) - Description

Returns audio status and the starting and ending locations of the last playback or the next resume position.

If a Stop Audio function (Category 81h, Function 51h) was issued to stop playing, the paused bit is set to 1; the starting location is where the playback was stopped, and the ending location should be the ending location that was specified in the last Play Audio command. If there was no preceding Play Audio or Stop Audio command, the paused bit should be set to 0, and the starting and ending locations should be the CD Redbook equivalents of those specified in the last Play Audio command.

If a Play Audio command has been completed (without being stopped), the paused bit should be set to 0, and the starting and ending locations should be as specified in the Play Audio command. If a Resume Audio function has been completed, the paused bit should be set to 0, and the starting and ending locations should be as they were at the beginning of the Resume Audio command.

Note: The addresses returned here are always in the format of addresses minutes/seconds/frames.

This command is performed regardless of whether the drive is already playing.

CDROMAUDIO_GETAUDIOSTATUS (65h) - Signature

Signature
Signature string of "CD01". The string is not NULL terminated.

CDROMAUDIO_GETAUDIOSTATUS (65h) - Parameter Packet For

Field	Length	C Datatype
Signature	4 BYTES	UCHAR[4]

CDROMAUDIO_GETAUDIOSTATUS (65h) - Audio status bits

Audio status bits
Audio status bits defined as follows:
00 = Audio paused bit
01-15 = Reserved

CDROMAUDIO_GETAUDIOSTATUS (65h) - Starting Location

Starting Location
Starting location in the MSF format of last Play Audio or next Resume Audio command.

CDROMAUDIO_GETAUDIOSTATUS (65h) - Ending Location

Ending Location
Ending Location in the MSF format of last Play Audio or next Resume Audio command.

CDROMAUDIO_GETAUDIOSTATUS (65h) - Data Packet Format

Field	Length	C Datatype
Audio status bits	WORD	USHORT
Starting Location	DWORD	ULONG
Ending Location	DWORD	ULONG

CDROMAUDIO_GETAUDIOSTATUS (65h) - Returns

The error return codes for this function are as follows:

10h	Uncertain media.
13h	Unsupported parameter. The first 4 bytes of the parameter buffer are not "CD01".

CDROMAUDIO_GETAUDIOSTATUS (65h) - Remarks

None.

CDROMAUDIO_GETAUDIOSTATUS (65h) -

Category:
IOCTL_CDROMAUDIO (81h)
Function:
CDROMAUDIO_GETAUDIOSTATUS (65h)
Description:
Return Audio-Status Information

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 81h Touch Device-Dependent Driver

All Touch device driver IOCtl commands share Category 81h commands, which are distinguished by the device name used in the device Open (PDITOU$ for the device-dependent driver, TOUCH$ for the device-independent driver).

Device-Dependent Device Driver Command Summary

The following table describes the Category 81h Touch Device-Dependent Driver IOCtl commands:

Function	Description
50h	Reserved.
51h	Reserved.
52h	Set Calibration Constants
53h	Read Data
54h	Set Data Mode
55h	Set Click-Lock Parameters
56h	Set Touch Thresholds
57h	Set Emulation XY Offset
58h	Set Data Report Rate
59h	Set Low Pass Filter
5Ah	Write Memory Location
5Bh	Reserved.
5Ch	Reserved.
5Dh	Reserved.
5Eh	Reserved.
5Fh	Reserved.
60h	Get Calibration Constants
61h	Get Data Mode
62h	Get Click-Lock Parameters
63h	Get Touch Thresholds
64h	Get Emulation XY Offset
65h	Get Data Report Rate
66h	Get Low Pass Filter
67h	Read Memory Location

TOUCH_DEVDEP_SETCALIBCONST (52h) - Set Calibration Const

TOUCH_DEVDEP_SETCALIBCONST (52h) - Description

This function downloads the Touch Display calibration constants.

TOUCH_DEVDEP_SETCALIBCONST (52h) - Parameter Packet Fo

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Channel 1 gain constant	DWORD	ULONG
Channel 2 gain constant	DWORD	ULONG
Channel 3 gain constant	DWORD	ULONG
Channel 4 gain constant	DWORD	ULONG
X as a function of X C[xx]	DWORD	ULONG
X as a function of Y C[xy]	DWORD	ULONG
Y as a function of X C[yx]	DWORD	ULONG
Y as a function of Y C[yy]	DWORD	ULONG
X offset C[x0]	DWORD	ULONG
Y offset C[y0]	DWORD	ULONG

TOUCH_DEVDEP_SETCALIBCONST (52h) - Data Packet Format

None.

TOUCH_DEVDEP_SETCALIBCONST (52h) - Returns

None.

TOUCH_DEVDEP_SETCALIBCONST (52h) - Remarks

This function is intended for the exclusive use of the Calibration utility.

All parameters in the Parameter Packet Format are in double-word, floating point format. This format is non-IEEE standard and is described in the *IBM Personal System/2 8516 Touch Display Technical Reference* .

TOUCH_DEVDEP_SETCALIBCONST (52h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETCALIBCONST (52h)
Description:
Set Calibration Constants

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_READDATA (53h) - Read Data

TOUCH_DEVDEP_READDATA (53h) - Description

Reads a single packet of data from the Touch Display controller. The format of the data, XYZ or Raw T1-4, depends on the data mode set using Function 54h.

TOUCH_DEVDEP_READDATA (53h) - Read type

Read type

Read type is used to determine the type of action taken if no event queue data is available. The values can be:

0	Return NULL EventInfo if data not available.
1	WAIT for data on empty queue.

All other values are reserved.

TOUCH_DEVDEP_READDATA (53h) - Parameter Packet Format

The parameter packet is a location in application storage where the caller indicates the action to be taken if no event queue data is available. The format of the parameter packet is as follows:

Field	Length	C Datatype
Read type	WORD	USHORT

TOUCH_DEVDEP_READDATA (53h) - Status

Status

Has the following bit-level definitions for XYZ mode:

Bit	*Description*
0	1=Hardware offset adjust error
1	1=Offset adjust error
2	1=Floating point calculation error
3	Reserved=0
4	1=Channel pegged error
5	1=Offset adjust just occurred
6	Reserved=0
7	Reserved=0
8	Reserved=0
9	Reserved=0
A	Reserved=0
B	Reserved=0
C	Touch screen status 2
D	Touch screen status 1
E	1=Selection detection on
F	0=XYZ packet

TOUCH_DEVDEP_READDATA (53h) - Time

Time
Time stamp in milliseconds.

TOUCH_DEVDEP_READDATA (53h) - X value

X value
Touch contact x (horizontal) position.

TOUCH_DEVDEP_READDATA (53h) - Y value

Y value
Touch contact y (vertical) position.

TOUCH_DEVDEP_READDATA (53h) - Z value

Z value
Proportional to the touch contact force.

TOUCH_DEVDEP_READDATA (53h) - Reserved

Reserved
Reserved.

TOUCH_DEVDEP_READDATA (53h) - Data Packet Format

The data packet is a location in application storage where the Touch display device driver returns to the caller a Touch display event queue record. This record has two formats, depending on the data type specified, using Function 54h. The two types of record are Raw data (see TOUCH_DEVDEP_READDATA) or XYZ data. The default setting depends on which mode was set using Function 54H. The format for the XYZ data mode is as follows:

Field	Length	C Datatype
Status	WORD	USHORT
Time	DWORD	ULONG
X value	WORD	USHORT
Y value	WORD	USHORT
Z value	WORD	USHORT
Reserved	WORD	USHORT

TOUCH_DEVDEP_READDATA (53h) - Returns

This function returns to the caller a touch event from the touch event (FIFO) queue flags.

TOUCH_DEVDEP_READDATA (53h) - Remarks

None.

TOUCH_DEVDEP_READDATA (53h) -

Category:

IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_READDATA (53h)
Description:
Read Data

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_READDATA (53h) - Read Data

TOUCH_DEVDEP_READDATA (53h) - Description

Reads a single packet of data from the Touch Display controller. The format of the data, XYZ or Raw T1-4, depends on the data mode set using Function 54h.

TOUCH_DEVDEP_READDATA (53h) - Read type

Read type

Read type is used to determine the type of action taken if no event queue data is available. The values can be:

0	Return NULL EventInfo if data not available.
1	WAIT for data on empty queue.

All other values are reserved.

TOUCH_DEVDEP_READDATA (53h) - Parameter Packet Format

The parameter packet is a location in application storage where the caller indicates the action to be taken if no event queue data is available. The format of the parameter packet is as follows:

Field	Length	C Datatype
Read type	WORD	USHORT

TOUCH_DEVDEP_READDATA (53h) - Status

Status

Has the following bit-level definitions for RAW mode:

Bit	*Description*
1	Reserved=0
2	Reserved=0
3	Reserved=0
4	Reserved=0
5	Reserved=0
6	1=Channel pegged
7	Reserved=0
8	Reserved=0
9	Reserved=0
A	1=Adjust just occurred
B	Reserved=0
C	Reserved=0
D	Reserved=0
E	1=Offset error
F	1=RAW data

TOUCH_DEVDEP_READDATA (53h) - Time

Time

Time stamp in milliseconds.

TOUCH_DEVDEP_READDATA (53h) - Transducer 1

Transducer 1

Raw transducer output value. Transducers are numbered clockwise from top left.

TOUCH_DEVDEP_READDATA (53h) - Transducer 2

Transducer 2

Raw transducer output value. Transducers are numbered clockwise from top left.

TOUCH_DEVDEP_READDATA (53h) - Transducer 3

Transducer 3
Raw transducer output value. Transducers are numbered clockwise from top left.

TOUCH_DEVDEP_READDATA (53h) - Transducer 4

Transducer 4
Raw transducer output value. Transducers are numbered clockwise from top left.

TOUCH_DEVDEP_READDATA (53h) - Data Packet Format

The data packet is a location in application storage where the Touch display device driver returns to the caller a Touch display event queue record. This record has two formats, depending on the data type specified, using Function 54H. The two types of record are XYZ data (see TOUCH_DEVDEP_READDATA) or Raw data. The default setting depends on which mode was set using Function 54h. The format for the Raw data packet is as follows:

Field	Length	C Datatype
Status	WORD	USHORT
Time	DWORD	ULONG
Transducer 1	WORD	USHORT
Transducer 2	WORD	USHORT
Transducer 3	WORD	USHORT
Transducer 4	WORD	USHORT

TOUCH_DEVDEP_READDATA (53h) - Returns

This function returns to the caller a touch event from the touch event (FIFO) queue flags.

TOUCH_DEVDEP_READDATA (53h) - Remarks

None.

TOUCH_DEVDEP_READDATA (53h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_READDATA (53h)
Description:
Read Data

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_SETDATAMODE (54h) - Set Data Mode

TOUCH_DEVDEP_SETDATAMODE (54h) - Description

Causes the Touch display to report either XYZ data or *raw T1-T4* strain gauge data.

TOUCH_DEVDEP_SETDATAMODE (54h) - Mode

Mode

Selects the relevant data packet reporting mode, as follows:

0	XYZ mode
1	RAW mode

TOUCH_DEVDEP_SETDATAMODE (54h) - Parameter Packet Forr

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Mode	DWORD	ULONG

TOUCH_DEVDEP_SETDATAMODE (54h) - Data Packet Format

See TOUCH_DEVDEP_READDATA for data packet formats.

TOUCH_DEVDEP_SETDATAMODE (54h) - Returns

None.

TOUCH_DEVDEP_SETDATAMODE (54h) - Remarks

RAW mode is intended for exclusive use by the Calibration utility.

TOUCH_DEVDEP_SETDATAMODE (54h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETDATAMODE (54h)
Description:
Set Data Mode

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Set Click-Lock Parame

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Description

Specifies the Touch Display click-lock parameters.

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Active Area

Active Area
Specifies the X, Y pel area, within a defined X, Y coordinate of first selection, in which a mouse emulation button 1 click (push harder threshold) will be held.

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Timeout Period

Timeout Period
Specifies, in milliseconds, the length of time a mouse emulation button 1 click will be held within a defined active area.

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Click Count

Click Count
Specifies that the mouse emulation button 1 click will be held within a defined area for a defined period of time. The values are:

0	Click-locking off (default)
1	Single-button click-locking
2	Double-button click-locking

Note: Any parameter value of -1 causes the driver default value to be substituted.

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Parameter Packet Forr

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Active Area	WORD	USHORT
Timeout Period	WORD	USHORT
Click Count	WORD	SHORT

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Data Packet Format

None.

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Returns

None.

TOUCH_DEVDEP_SETCLICKLOCK (55h) - Remarks

This function is particularly useful for manipulating small areas using mouse emulation (or example, moving Presentation Manager icons). It is used by the Touch Control Panel utility.

TOUCH_DEVDEP_SETCLICKLOCK (55h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETCLICKLOCK (55h)
Description:
Set Click-Lock Parameters

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Set Touch Th

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Description

Specifies the Touch Display thresholds.

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - On-Screen th

On-Screen threshold
Sets the value of Z, on the chosen Z scale, at which the Touch display interprets the touch contact force as *On-screen* and starts reporting data. Data is reported only while the force exceeds this threshold.

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Push Thresho

Push Threshold
Sets the value of Z, when Z is increasing on the chosen Z scale, at which the Touch display sets the selection detection bit in the status returned by Function 53h.

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Push hysteres

Push hysteresis threshold

Sets the value of Z, when Z is decreasing on the chosen Z scale, at which the Touch display resets the selection detection bit in the status returned by Function 53h.

Note: Any parameter value of -1 causes the driver default value to be substituted.

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Parameter Pa

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
On-Screen threshold	WORD	USHORT
Push Threshold	WORD	USHORT
Push hysteresis threshold	WORD	SHORT

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Data Packet F

None.

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Returns

None.

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) - Remarks

Push and push hysteresis thresholds are relevant only when the selection detection mechanism of *push harder* is set. (See TOUCH_DEVINDEP_SETSELECTMECH.)

TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETTOUCHTHRESHOLD (56h)
Description:
Set Touch Thresholds

Select an item:

Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_SETEMULXY (57h) - Set Emulation XY Offset

TOUCH_DEVDEP_SETEMULXY (57h) - Description

Specifies the Touch Display XY offset applied to touch data when it is passed as emulated mouse data.

TOUCH_DEVDEP_SETEMULXY (57h) - X offset

X offset
Indicates the amount by which the X coordinate is offset.

TOUCH_DEVDEP_SETEMULXY (57h) - Y offset

Y offset
Indicates the amount by which the Y coordinate is offset.

TOUCH_DEVDEP_SETEMULXY (57h) - Parameter Packet Format

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
X offset	WORD	SHORT
Y offset	WORD	SHORT

TOUCH_DEVDEP_SETEMULXY (57h) - Data Packet Format

None.

TOUCH_DEVDEP_SETEMULXY (57h) - Returns

None.

TOUCH_DEVDEP_SETEMULXY (57h) - Remarks

Calls to the mouse device-independent driver by way of the IDC interface will have the X and Y coordinate offsets applied to the touch X, Y, Z data. Data supplied to the Touch device-independent driver is not affected.

TOUCH_DEVDEP_SETEMULXY (57h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETEMULXY (57h)
Description:
Set Emulation XY Offset

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Set Data Repo

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Description

Specifies the Touch Display data report rate.

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Report rate

Report rate
Sets the maximum rate at which data packets are returned by the Touch display while the touch contact force exceeds the On-Screen threshold (see TOUCH_DEVDEP_SETTOUCHTHRESHOLD.) The values are:

0	10 reports per second
1	20 reports per second
2	30 reports per second
3	40 reports per second
4	60 reports per second
-1	Driver default value

All other values are reserved.

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Parameter Pa

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Report rate	WORD	SHORT

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Data Packet F

None.

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Returns

None.

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) - Remarks

None.

TOUCH_DEVDEP_SETDATAREPORTRATE (58h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETDATAREPORTRATE (58h)
Description:
Set Data Report Rate

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Set Low-Pass Fi

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Description

Specifies the Touch Display low pass filter parameters.

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Filter type

Filter type

Determines the type of low pass filter to be used:

0	Default (Touch Display)
1	Alternative (Touch Panel)

All other values are reserved.

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Filter frequency

Filter frequency

Determines the frequency cut-off value for the filter:

0	Default (Medium)
1	Slow
2	Medium
3	Fast
4	Rapid

All other values are reserved.

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Parameter Pack

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Filter type	USHORT	USHORT
Filter frequency	USHORT	USHORT

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Data Packet For

None.

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Returns

None.

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) - Remarks

This function is used by the Touch Control Panel.

TOUCH_DEVDEP_SETLOWPASSFILTER (59h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_SETLOWPASSFILTER (59h)
Description:
Set Low-Pass Filter

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) - Write Memory Locatio

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) - Description

Writes a byte to Touch Display controller RAM.

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) - Parameter Packet For

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Data to write	BYTE	UCHAR
Address to write at	WORD	USHORT

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) - Data Packet Format

None.

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) - Returns

None.

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) - Remarks

None.

TOUCH_DEVDEP_WRITEMEMLOC (5Ah) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_WRITEMEMLOC (5Ah)
Description:
Write Memory Location

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETCALIBCONST (60h) - Get Calibration Cons

TOUCH_DEVDEP_GETCALIBCONST (60h) - Description

Gets the Touch Display calibration constants.

TOUCH_DEVDEP_GETCALIBCONST (60h) - Parameter Packet Fo

None.

TOUCH_DEVDEP_GETCALIBCONST (60h) - Data Packet Format

Field	Length	C Datatype
Channel 1 gain constant	DWORD	ULONG
Channel 2 gain constant	DWORD	ULONG
Channel 3 gain constant	DWORD	ULONG
Channel 4 gain constant	DWORD	ULONG
X as a function of X C[xx]	DWORD	ULONG
X as a function of Y C[xy]	DWORD	ULONG
Y as a function of X C[yx]	DWORD	ULONG
Y as a function of Y C[yy]	DWORD	ULONG
X offset C[x0]	DWORD	ULONG
Y offset C[y0]	DWORD	ULONG

TOUCH_DEVDEP_GETCALIBCONST (60h) - Returns

None.

TOUCH_DEVDEP_GETCALIBCONST (60h) - Remarks

This function is intended for exclusive use by the Calibration utility.

All parameters are in double-word floating-point format. This format is non-IEEE standard and is described in the *IBM Personal System/2 8516 Touch Display Technical Reference* .

TOUCH_DEVDEP_GETCALIBCONST (60h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETCALIBCONST (60h)
Description:
Get Calibration Constants

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETDATAMODE (61h) - Get Data Mode

TOUCH_DEVDEP_GETDATAMODE (61h) - Description

Gets the current Touch Display data packet reporting mode.

TOUCH_DEVDEP_GETDATAMODE (61h) - Parameter Packet For

None.

TOUCH_DEVDEP_GETDATAMODE (61h) - Mode

Mode

Indicates the current data packet reporting mode:

0	XYZ mode
1	RAW mode

See Function 53h for the data packet formats.

TOUCH_DEVDEP_GETDATAMODE (61h) - Data Packet Format

The data packet is a location in application storage where the Touch Display driver returns to the caller the current data packet reporting mode. The format is as follows:

Field	Length	C Datatype
Mode	WORD	USHORT

TOUCH_DEVDEP_GETDATAMODE (61h) - Returns

None.

TOUCH_DEVDEP_GETDATAMODE (61h) - Remarks

This function returns to the caller the current Touch Display data packet reporting mode.

TOUCH_DEVDEP_GETDATAMODE (61h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETDATAMODE (61h)
Description:
Get Data Mode

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Get Click-Lock Parame

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Description

Gets the Touch Display click-lock parameters.

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Parameter Packet For

None.

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Active area

Active area
Specifies the x,y pel area, within a defined x,y coordinate of first selection, in which a mouse emulation button 1 click will be held.

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Timeout period

Timeout period
Specifies, in milliseconds, the length of time a mouse emulation button 1 click will be held within a defined active area.

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Click count

Click count
Specifies that the mouse emulation button 1 click will be held within a defined area for a defined period of time. Values are:

0	Click-locking off (default)
1	Single-button click-locking
2	Double-button click-locking

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Data Packet Format

The data packet is a location in application storage where the Touch Display driver returns to the caller the current click-lock settings. The format is as follows:

Field	Length	C Datatype
Active area	WORD	USHORT
Timeout period	WORD	USHORT
Click count	WORD	USHORT

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Returns

This function returns to the caller the mouse emulation click-lock settings.

TOUCH_DEVDEP_GETCLICKLOCK (62h) - Remarks

This function is particularly useful for manipulating small areas using mouse emulation (for example moving Presentation Manager icons).

The tendency of the emulated mouse position to change while a button click is being attempted can be eliminated with only a small initial hesitation, noted as a side effect, while the user attempts to drag the cursor.

A setting in the range of 3-500 milliseconds for the timeout period is recommended, while the active area setting will depend on the screen pel resolution. As a guide, use a value approximately equal to half the size of an icon.

TOUCH_DEVDEP_GETCLICKLOCK (62h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETCLICKLOCK (62h)
Description:
Get Click-Lock Parameters

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Get Touch Th

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Description

Gets the current Touch Display thresholds.

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Parameter Pa

None.

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - On-Screen th

On-Screen threshold

Is the value of Z, on the chosen Z scale, at which the Touch display interprets the touch contact force as *On-screen* and starts reporting data. Data is reported only while the force exceeds this threshold.

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Push Thresh

Push Threshold

Is the value of Z, when Z is increasing on the chosen Z scale, at which the Touch display sets the selection detection bit in the status returned by Function 53h.

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Push hystere

Push hysteresis threshold

Is the value of Z, when Z is decreasing on the chosen Z scale, at which the Touch display resets the selection detection bit in the status returned by Function 53h.

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Data Packet

The data packet is a location in application storage where the Touch Display driver will return to the caller the current Touch Display threshold settings. The format is as follows:

Field	Length	C Datatype
On-Screen threshold	WORD	USHORT
Push Threshold	WORD	USHORT
Push hysteresis threshold	WORD	USHORT

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Returns

This function returns to the caller the current threshold values.

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) - Remarks

Push and push hysteresis thresholds are relevant only when the selection detection mechanism of *push harder* is set. (See TOUCH_DEVINDEP_SETSELECTMECH.)

TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETTOUCHTHRESHOLD (63h)
Description:
Get Touch Thresholds

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETEMULXY (64h) - Get Emulation XY Offset

TOUCH_DEVDEP_GETEMULXY (64h) - Description

Gets the Touch Display XY offsets applied during mouse emulation.

TOUCH_DEVDEP_GETEMULXY (64h) - Parameter Packet Format

None.

TOUCH_DEVDEP_GETEMULXY (64h) - X offset

X offset

Indicates the amount by which the X coordinate will be offset.

TOUCH_DEVDEP_GETEMULXY (64h) - Y offset

Y offset

Indicates the amount by which the Y coordinate will be offset.

TOUCH_DEVDEP_GETEMULXY (64h) - Data Packet Format

The data packet is a location in application storage where the Touch Display driver returns to the caller the values of the X and Y offsets, which are added to the basic X,Y data before being supplied to the Mouse device-independent driver by way of its absolute IDC entry point. The format is as follows:

Field	Length	C Datatype
X offset	WORD	SHORT
Y offset	WORD	SHORT

TOUCH_DEVDEP_GETEMULXY (64h) - Returns

This function returns to the caller the current X and Y coordinate offsets applied during mouse emulation.

TOUCH_DEVDEP_GETEMULXY (64h) - Remarks

This function is used by the Touch Control Panel to offset the mouse pointer image from the finger, so that the image can be seen while the user touches the screen.

TOUCH_DEVDEP_GETEMULXY (64h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETEMULXY (64h)
Description:
Get Emulation XY Offset

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Get Data Rep

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Description

Gets the current Touch Display data report rate.

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Parameter Pa

None.

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Report rate

Report rate

Is the maximum rate at which data packets will be returned by the Touch display while the touch contact force exceeds the On-Screen threshold (see TOUCH_DEVDEP_SETTOUCHTHRESHOLD.) The values are:

0	10 reports per second
1	20 reports per second
2	30 reports per second
3	40 reports per second
4	60 reports per second
-1	Driver default value

All other values are reserved.

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Data Packet F

The data packet is a location in application storage where the Touch Display driver returns to the caller the Touch Display maximum data reporting rate. The format is as follows:

Field	Length	C Datatype
Report rate	WORD	SHORT

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Returns

This function returns to the caller the data report rate.

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) - Remarks

The actual reporting rate might vary up to the maximum quoted rate.

TOUCH_DEVDEP_GETDATAREPORTRATE (65h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETDATAREPORTRATE (65h)
Description:
Get Data Report Rate

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Get Low-Pass F

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Description

Gets the current Touch Display low-pass filter parameters.

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Parameter Pack

None.

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Filter type

Filter type
Indicates the type of low pass filter to be used:

0	Default (Touch Display)
1	Alternative (Touch Panel)

All other values are reserved.

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Filter frequency

Filter frequency

Indicates the frequency cut-off value for the filter:

0	Default (Medium)
1	Slow
2	Medium
3	Fast
4	Rapid

All other values are reserved.

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Data Packet For

The data packet is a location in application storage where the Touch Display driver returns to the caller the low-pass filter settings. The format is as follows:

Field	Length	C Datatype
Filter type	USHORT	USHORT
Filter frequency	USHORT	USHORT

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Returns

This function returns to the caller the low-pass filter settings.

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) - Remarks

A single set of filter constants, set in one screen group, applies screen-group wide. The Touch Control Panel uses this function.

TOUCH_DEVDEP_GETLOWPASSFILTER (66h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_GETLOWPASSFILTER (66h)
Description:
Get Low-Pass Filter

Select an item:
Description
Parameter Packet Format
Data Packet Format

Returns
Remarks

TOUCH_DEVDEP_READMEMLOC (67h) - Read Memory Location

TOUCH_DEVDEP_READMEMLOC (67h) - Description

Reads a byte from Touch Display controller RAM.

TOUCH_DEVDEP_READMEMLOC (67h) - Parameter Packet Form

None.

TOUCH_DEVDEP_READMEMLOC (67h) - Data Packet Format

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Data read from RAM	BYTE	UCHAR
Address to read from	WORD	USHORT

TOUCH_DEVDEP_READMEMLOC (67h) - Returns

None.

TOUCH_DEVDEP_READMEMLOC (67h) - Remarks

None.

TOUCH_DEVDEP_READMEMLOC (67h) -

Category:
IOCTL_TOUCH_DEVDEP (81h)
Function:
TOUCH_DEVDEP_READMEMLOC (67h)
Description:
Read Memory Location

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Category 81h Touch Device-Independent Driver

All Touch device driver IOCtl commands share Category 81h commands, which are distinguished by the device name used in the device Open (PDITOU$ for the device-dependent driver, TOUCH$ for the device-independent driver).

Device-Independent Device Driver Command Summary

The following lists and describes the Category 81h Touch Device-Independent Driver:

Function	Description
50h	Set Coordinate System
51h	Reserved
52h	Set Selection Mechanism
53h	Set Event Mask
54h	Set Queue Size
55h	Set Emulation State
60h	Get Coordinate System
61h	Reserved
62h	Get Selection Mechanism
63h	Get Event Mask
64h	Get Queue Size
65h	Get Emulation State
66h	Get Read Event Queue

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Set Coordinate Syst

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Description

Specifies the Touch Display coordinate system.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - X min

X min

The value of X to be reported at the left-hand edge of the display.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - X max

X max

The value of X to be reported at the right-hand edge of the display.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Y min

Y min

The value of Y to be reported at the top edge of the display.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Y max

Y max

The value of Y to be reported at the bottom edge of the display.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Z min

Z min

The notational value that would be reported with minimum contact force applied. Z values are reported only when they exceed the On-Screen threshold. Values from Z min to On-Screen threshold are reported.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Z max

Z max
The value to be reported with maximum contact force applied.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - X hysteresis

X hysteresis
The amount of hysteresis in the horizontal direction in cell units multiplied by 1000. For example, hysteresis of 0.4 returns 400.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Y hysteresis

Y hysteresis
The amount of hysteresis in the vertical direction in cell units multipled by 1000. For example, hysteresis of 0.4 returns 400.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Parameter Packet F

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
X min	WORD	USHORT
X max	WORD	USHORT
Y min	WORD	USHORT
Y max	WORD	USHORT
Z min	WORD	USHORT
Z max	WORD	USHORT
X hysteresis	WORD	USHORT
Y hysteresis	WORD	USHORT

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Data Packet Format

None.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Returns

None.

TOUCH_DEVINDEP_SETCOORDSYS (50h) - Remarks

X and Y values are treated as 16-bit 2's complement signed integers in the range -32768 to 32767.

Hysteresis values are unsigned integers in the range 0 - 65535.

Hysteresis values required should be expressed in terms of the hysteresis value (in X, Y coordinate system units) multiplied by 1000.

TOUCH_DEVINDEP_SETCOORDSYS (50h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_SETCOORDSYS (50h)
Description:
Set Coordinate System

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Set Selection Mec

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Description

Specifies the Touch Display selection mechanism method.

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Type

Type

The field in which the selection detection flag (bit) in the status word returned in Function 66h is set or reset:

Bit 0	
0	Push harder (default)
1	Lift off
Bit 1	
0	First value taken
1	Single-touch algorithm applied
Bit 2	

0	First value taken
1	Stable-region algorithm active

Bit 3-F

0	Reserved.

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Number of stable

Number of stable points

0	Default
n	Number of stable points

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Tolerance value

Tolerance value

0	Default
n	Tolerance in pels

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Stable region leng

Stable region length

0	Default
n	Region length in samples (0 < n < 60)

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Parameter Packet

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Type	WORD	USHORT
Number of stable points	WORD	USHORT
Tolerance value	WORD	USHORT
Stable region length	WORD	USHORT

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Data Packet Form

None.

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Returns

None.

TOUCH_DEVINDEP_SETSELECTMECH (52h) - Remarks

Information on the single-touch and stable-region algorithms is shown in the *Input/Output Driver Reference* .

TOUCH_DEVINDEP_SETSELECTMECH (52h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_SETSELECTMECH (52h)
Description:
Set Selection Mechanism

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Set Event Mask

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Description

Assigns a new Touch Display event mask.

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Event Mask

Event Mask

Where:

0	1=On-Screen status set event.
1	1=Touch-Down event (On-Screen status clear to set)
2	1=Lift-Off event (On-Screen status set to clear)
3	1=Selection Detection flag set event
4	1=Selection Detection flag clear to set event
5	1=Selection Detection flag set to clear event
6-F	Reserved=0

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Parameter Packet F

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Event Mask	WORD	USHORT

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Data Packet Forma

None.

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Returns

None.

TOUCH_DEVINDEP_SETEVENTMASK (53h) - Remarks

The Touch Display device driver gets the new mask for enabled/disabled Touch Display device events from the caller's parameter packet. The mask determines which events are to be queued (that is, *readable)* by calling TOUCH_DEVDEP_GETLOWPASSFILTER.

TOUCH_DEVINDEP_SETEVENTMASK (53h) -

Category:

IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_SETEVENTMASK (53h)
Description:
Set Event Mask

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Set Queue Size

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Description

Sets the Touch Display event queue size.

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Queue Size

Queue Size
Indicates how many past Touch Display events will be stored in the event queue.

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Parameter Packet F

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Queue Size	WORD	USHORT

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Data Packet Format

None.

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Returns

None.

TOUCH_DEVINDEP_SETQUEUESIZE (54h) - Remarks

The queue size set can be any value from 1 to a maximum of the QSIZE parameter specified in the device driver CONFIG.SYS statement (which defaults to 10 if not specified).

TOUCH_DEVINDEP_SETQUEUESIZE (54h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_SETQUEUESIZE (54h)
Description:
Set Queue Size

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Set Emulation State

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Description

Specifies the Touch Display mouse emulation state: On or Off.

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Emul State

Emul State

Switch that turns mouse emulation On or Off for the screen group.

Bit	*Description*
0	Screen group emulation 0=Emulation off; 1=Emulation on
1-F	Reserved

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Parameter Packet F

The parameter packet is a location in application storage that contains the following data:

Field	Length	C Datatype
Emul State	WORD	USHORT

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Data Packet Forma

None.

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Returns

None.

TOUCH_DEVINDEP_SETEMULSTATE (55h) - Remarks

This function calls the device-dependent driver by way of the IDC to set or reset the emulation state.

Using this function, emulation can be turned On after the Touch device is opened (which disables emulation temporarily), assuming that emulation is available with EMUL=1 specified in the Touch Display TOUCH.INI configuration file.

TOUCH_DEVINDEP_SETEMULSTATE (55h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_SETEMULSTATE (55h)
Description:
Set Emulation State

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Get Coordinate Syst

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Description

Gets the Touch Display coordinate system.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Parameter Packet F

None.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - X min

X min
The value of X to be reported at the left-hand edge of the display.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - X max

X max
The value of X to be reported at the right-hand edge of the display.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Y min

Y min
The value of Y to be reported at the top edge of the display.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Y max

Y max
The value of Y to be reported at the bottom edge of the display.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Z min

Z min

The notation value that would be reported with minimal contact force applied. Z values are reported only when they exceed the On-Screen threshold. Values from Z min to On-Screen threshold are reported.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Z max

Z max

The value to be reported with maximum contact force applied.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - X hys

X hys

The amount of hysteresis in the horizontal direction in cell units multiplied by 1000. For example, hysteresis of 0.4 returns 400.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Y hys

Y hys

The amount of hysteresis in the vertical direction in cell units multiplied by 1000. For example, hysteresis of 0.4 returns 400.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Data Packet Format

The data packet is a location in application storage where the Touch Display driver returns to the caller the coordinate system being used to report X,Y,Z data report values, including maximum and minimum limits and change report hysteresis values. The format is as follows:

Field	Length	C Datatype
X min	WORD	USHORT
X max	WORD	USHORT
Y min	WORD	USHORT
Y max	WORD	USHORT
Z min	WORD	USHORT
Z max	WORD	USHORT
X hys	WORD	USHORT
Y hys	WORD	USHORT

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Returns

This function returns to the caller the current X,Y,Z coordinate reporting range and the X,Y coordinate change hysteresis values.

TOUCH_DEVINDEP_GETCOORDSYS (60h) - Remarks

X and Y values are 16-bit 2's complement signed integers in the range -32768 to 32767.

Hysteresis values are unsigned integers in the range 0 - 65535.

Hysteresis values are expressed in terms of the hysteresis value (in X,Y coordinate system units) multiplied by 1000.

TOUCH_DEVINDEP_GETCOORDSYS (60h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_GETCOORDSYS (60h)
Description:
Get Coordinate System

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Get Selection Me

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Description

Gets the Touch Display selection mechanism method.

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Parameter Packe

None.

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Type

Type

The field in which the selection detection flat (bit) in the status word returned in Function 66h is set or reset.

Bit 0	
0	Push harder (default)
1	Lift off
Bit 1	
0	First value taken
1	Single-touch algorithm applied
Bit 2	
0	First value taken
1	Stable-region algorithm active
Bit 3-F	
0	Reserved.

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Number of stable

Number of stable points

0	Default
n	Number of stable points

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Tolerance value

Tolerance value

0	Default
n	Tolerance in pels

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Stable region leng

Stable region length

0	Default
n	Region length in samples (0 < n < 60)

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Data Packet Form

The data packet is a location in application storage where the Touch Display driver returns to the caller the Touch Display selection mechanism method. The format is as follows:

Field	Length	C Datatype
Type	WORD	USHORT
Number of stable points	WORD	USHORT
Tolerance value	WORD	USHORT
Stable region length	WORD	USHORT

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Returns

None.

TOUCH_DEVINDEP_GETSELECTMECH (62h) - Remarks

Information on the single-touch and stable-region algorithms is shown in the *Input/Output Driver Reference* .

TOUCH_DEVINDEP_GETSELECTMECH (62h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_GETSELECTMECH (62h)
Description:
Get Selection Mechanism

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Get Event Mask

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Description

Gets the current Touch Display event mask.

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Parameter Packet I

None.

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Event Mask

Event Mask

Where:

Bit	*Meaning*
0	1=On-Screen status set event
1	1=Touch-Down event (On-Screen status clear to set)
2	1=Lift-Off event (On-Screen status set to clear)
3	1=Selection Detection flag set event
4	1=Selection Detection flag clear to set event
5	1=Selection Detection flag set to clear event
6-F	0=Reserved

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Data Packet Forma

The data packet is a location in application storage where the Touch Display driver will return to the caller the Touch Display event mask, which controls the type of events that are to be added to the event queue. The format is as follows:

Field	Length	C Datatype
Event Mask	WORD	USHORT

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Returns

None.

TOUCH_DEVINDEP_GETEVENTMASK (63h) - Remarks

None.

TOUCH_DEVINDEP_GETEVENTMASK (63h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_GETEVENTMASK (63h)
Description:
Get Event Mask

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Get Queue Size

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Description

Gets the current Touch Display queue size.

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Parameter Packet F

None.

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Queue Size

Queue Size

Indicates how many previous Touch Display events are being stored in the event queue.

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Data Packet Format

The data packet is a location in application storage where the Touch Display driver will return to the caller the size of the Touch Display event queue. The format is as follows:

Field	Length	C Datatype
Queue Size	WORD	USHORT

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Returns

None.

TOUCH_DEVINDEP_GETQUEUESIZE (64h) - Remarks

None.

TOUCH_DEVINDEP_GETQUEUESIZE (64h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_GETQUEUESIZE (64h)
Description:
Get Queue Size

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Get Emulation State

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Description

Gets the current Touch Display mouse emulation state.

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Parameter Packet F

None.

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Emul State

Emul State

Switch that turns mouse emulation on or off for the screen group.

Bit	*Description*
Bit 0	Screen group emulation 0=Emulation off; 1=Emulation on
Bits 1-F	Reserved

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Data Packet Forma

The data packet is a location in application storage where the Touch Display driver will return the caller to the Touch Display mouse emulation state. The format is as follows:

Field	Length	C Datatype
Emul State	WORD	USHORT

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Returns

None.

TOUCH_DEVINDEP_GETEMULSTATE (65h) - Remarks

This function calls the device-dependent driver by way of the IDC to get the emulation state.

TOUCH_DEVINDEP_GETEMULSTATE (65h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_GETEMULSTATE (65h)
Description:
Get Emulation State

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Get Read E

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Description

Reads an event from the Touch Display event queue.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Parameter

None.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Remaining

Remaining Events
Count of the number of events remaining in the queue besides this current event. Less than or equal to queue size -1.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Status

Status
Status bits pertaining to this Touch Display event.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Time Stam

Time Stamp
Relative time at which the event occurred (in milliseconds).

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - X

X

Horizontal coordinate value of the touch event.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Y

Y

Vertical coordinate value of the touch event.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Z

Z

Third dimension coordinate value of the touch event.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Data Packe

The data packet is a location in application storage where the Touch Display driver will return to the caller the Touch Display event. The format is as follows:

Field	Length	C Datatype
Remaining Events	WORD	USHORT
Status	WORD	USHORT
Time Stamp	WORD	USHORT
X	WORD	USHORT
Y	WORD	USHORT
Z	WORD	USHORT

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Returns

None.

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) - Remarks

Bit definitions of Event Status are identical to those of the *Event Mask* emulation state. (See also TOUCH_DEVINDEP_SETEVENTMASK and TOUCH_DEVINDEP_GETEVENTMASK.)

TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h) -

Category:
IOCTL_TOUCH_DEVINDEP (81h)
Function:
TOUCH_DEVINDEP_GETREADEVENTQUEUE (66h)
Description:
Get Read Event Queue

Select an item:
Description
Parameter Packet Format
Data Packet Format
Returns
Remarks

Memory Management

This chapter describes the memory management features and functions of OS/2. The key features of OS/2 memory management are paged virtual memory and a 32-bit linear (flat) address space that is mapped through page tables to physical memory. An OS/2 application can allocate memory for its own use or to be shared with other applications.

The following topics are related to the information in this chapter:

- Exception handling
- Program execution and control
- Semaphores
- Queues

About Memory Management

OS/2 offers developers a 32-bit, linear (flat) memory address space. OS/2 uses a paged memory structure. OS/2 allocates, protects, and manipulates memory in terms of pages.

Process Address Space

The OS/2 memory allocation functions return a 32-bit pointer to the allocated memory object. While a 32-bit pointer is sufficient to address the entire 4 gigabyte global address space, applications can access only the first 512MB of linear memory, called the *process address space* . Of this 512MB process address space, a minimum of 64MB is reserved for shared memory regions, leaving 448MB. Of this 448MB, some will be used by the application itself and a small amount will be taken by operating system overhead. The remainder is available for allocation. The amount of memory that can actually be committed and used is, of course, determined by the physical memory and hard disk space available on the machine.

Keep in mind that the amount of memory that can be committed for actual use is limited by the amount of physical memory and free hard disk space that is available on the computer on which the application is executing.

Memory Objects

Applications allocate and manipulate memory in terms of *memory objects* . A memory object consists of one or more pages of memory. An OS/2 application can allocate any number of memory objects, within the following limits:

- the physical memory in the system
- the free hard disk space on the hard disk containing the swap file
- the 512MB process address space limit (see Process Address Space).

When requesting memory, the size of the memory object is rounded up to the next higher multiple of 4KB. An application can suballocate a memory object into memory blocks whose size can range from 1 byte to the size of the memory object.

Memory objects have the following characteristics:

- They are not relocatable.
- They are allocated in units of 4KB. One 4KB unit is called a *page* .
- They can be larger than 64KB in size.

Memory Pages

OS/2 allocates and commits memory objects in pages. A memory *page* is a 4KB (4096 bytes) piece of memory. Memory access protection is also done on a page-basis, rather than the segment-based protection used in previous versions of OS/2.

A *page range* is a linearly contiguous group of pages within a memory object. A page range can be any of the following:

- The entire memory object
- Part of the memory object
- A single page within a memory object

If an application requests 512 bytes of memory, it will receive a 32-bit pointer to a 4KB page. All 4096 bytes are available to the application, even though the request specified only 512 bytes. If an application requests 62000 bytes, it will receive a pointer to a 65536-byte (64KB, 16-page) object. Again, all 65536 bytes are available for use.

Each page in the virtual address space of the process is either free (unallocated), private (available only to the process that allocated it), or shared (memory that is shared between processes).

Each page within a memory object can be in one of two states, either uncommitted (that is, the linear address range has been reserved, but is not yet backed by physical storage) or committed (physical storage has been allotted for the logical address range).

Access to a committed page is controlled by the page's access protection attribute. These protection attributes are read access, write access, execute access (on the 80386, this is the same as read access), and guard page access.

An uncommitted page is not accessible.

Memory Overcommitment and Swapping

Memory overcommitment occurs when applications allocate and commit more memory than is actually available in the computer. OS/2 handles memory overcommitment by copying memory to the system swap file, SWAPPER.DAT, on the hard disk then reusing the memory for another allocation. OS/2 copies as many pages of memory as are necessary to make room for the new allocation. The swapped memory can be retrieved the next time it is accessed; at that time, some other memory might be written to the swap file.

OS/2 selects the memory to swap based on when it was last used. The page that is least-recently-used, that is, the page that has gone the longest since its last access, is the page chosen to swap to disk.

Swapping is transparent to an application, although excessive swapping can cause an application to run slowly.

Through swapping, OS/2 enables applications to allocate more memory than actually exists in the computer, bounded only by the amount of free space on the hard disk that contains the swap file.

User Configuration of Memory Swapping

Although an application cannot control swapping, the user can specify whether the system can swap memory by including the MEMMAN command in the CONFIG.SYS file.

If the MEMMAN command specifies SWAP, OS/2 writes selected memory pages to the SWAPPER.DAT file whenever insufficient physical memory exists to satisfy an allocation request. This is the default choice. If the MEMMAN command specifies NOSWAP, OS/2 does not swap memory.

Note: Be aware that disabling swapping will severely limit the number of applications that the user will be able to run concurrently; if there is not enough physical memory present, OS/2 might not even boot.

The exact amount of memory available to an application depends on the amount of physical memory in the machine and the amount of free disk space on the partition that contains the SWAPPER.DAT file. The location of the SWAPPER.DAT file can be specified by including the SWAPPATH command in the CONFIG.SYS file. With the SWAPPATH command, one can specify the location of the SWAPPER.DAT file, the amount of free disk space to reserve in the partition which will contain SWAPPER.DAT, and the initial size of SWAPPER.DAT. OS/2 adjusts the size of SWAPPER.DAT as necessary, leaving other files on the drive and the requested free space untouched.

Memory Allocation and Commitment

When an application asks OS/2 to allocate memory, a linear address range is reserved. The range is not backed by physical memory until the memory is *committed*. Commitment assigns physical memory to the linear address range.

A memory object that is allocated, but not committed is called a *sparse memory object*. A sparse memory object must be committed before it

can be used. An attempt to read from or write to uncommitted memory will cause an access violation.

An application can ask OS/2 to commit the memory at the same time it is allocated, thus making it immediately usable, or the memory can be committed at a later time. If the application commits the memory at the same time the memory is allocated, the entire memory object is committed. If the application commits the memory at a later time, it can commit the entire sparse memory object or only commit a portion of it.

When multiple pages are committed at the same time (a page range), the pages will have sequential linear addresses.

Managing Memory Allocation and Commitment
The recommended way to manage memory is to make a large memory allocation early in program execution, then to commit or suballocate memory as the need occurs.

The initial allocation should be for as much space as you expect to use during program execution. Allocation without commitment does not actually use any physical memory, so there is no waste involved in allocating several megabytes.

After the memory object is allocated, the application uses one of two ways to manage the memory object:

- Commit and decommit the memory as it is required
- Set up the memory object as a heap and suballocate memory from the heap.

Committing and decommitting memory gives the application more control over the process, but the application will have to keep track of which pages are committed and which pages are not. When suballocating memory from a heap, the application can have OS/2 track commitment and decommitment of physical memory pages, so the application does not have to. If you want DosSubAllocMem to manage the commitment of the pages spanned by the heap, all of the pages spanned by the memory object must be uncommitted initially.

Remember, no matter how much memory is originally allocated, the amount that an application will ultimately be able to commit and use is limited by the amount of physical memory and free disk space available on the machine.

Applications are not limited to a single large allocation of memory-other memory allocations can be made as necessary during execution-but large allocations and small commitments or suballocations are the most efficient way to manage memory.

Memory Resizing and Reallocation

In earlier versions of OS/2, an application could increase or decrease the size of an allocated memory segment by reallocating the segment.

Memory objects cannot be resized. Instead, an application should allocate a sparse memory object of whatever size might be necessary, then commit or decommit portions of the object.

If the amount of memory required cannot be determined at the time the memory is allocated, the application should allocate a sparse memory object large enough to meet the largest memory requirement. The application can then change the amount of committed memory as necessary.

For example, if you anticipate your application will use around 512KB of memory for most purposes, but might use 5MB under certain circumstances, you might take the following steps:

- During program initialization, use DosAllocMem to allocate 5MB.
- Commit the first 512KB (or some part of it) using DosSetMem.
- Proceed with normal processing.
- If extra memory is required occasionally, commit it and decommit it using DosSetMem.
- When the situation arises that the application requires the full 5MB, commit it at that time, using DosSetMem, then decommit it after you are finished with it, also using DosSetMem.
- When the application is finished with the memory, use DosFreeMem to release the memory back to the system.

Memory Protection

When an application allocates a memory object, it can specify the type of access to allow to the object. Memory access protection provides a program with control over the type of access that its threads have to a page of memory.

Access protection can only be defined for committed pages of memory and is initially set at the time the memory is committed. Different pages within the same memory object can have different access attributes and access attributes can be changed on a page-by-page basis at any

time.

An application can request any combination of the following access protection attributes:

Memory Access Protection Attributes

Access	Defined Constant	Description
Read Access	PAG_READ	The object can be read from, but not written to.
Write Access	PAG_WRITE	The object can be written to. On the 80386 microprocessor, write access implies both read and execute access.
Execute Access	PAG_EXECUTE	This is equivalent to read access on the 80386.
Guard Page Access	PAG_GUARD	Causes a guard-page-entered exception to be raised in a process that attempts to access the memory. This exception can be ignored or handled by the application's exception handler, if one is registered.

The guard page attribute is intended to provide automatic stack growth and stack limit checking. An application can also use it in other data structures, such as arrays. For example, if an application is using an array of 4096 bytes (one page), the application can allocate and commit two pages, one with read and write access and one designated as a guard page. If the application tries to write past the end of the array a page guard exception will be generated.

Any reference-read, write, or execute-to a guard page causes an access violation (page fault) to be generated. This fault causes a Guard Page Entered exception to occur for the thread that referred to the guard page. The exception can be handled by the exception handler of the process, if one is registered. If the process does not have an exception handler registered, OS/2's default exception handler will handle the exception. The default action by the system exception handler is to convert the page from a guard page to a committed page, then try to mark the next page in memory as a guard page. If the system is not successful in marking the next page as a guard page, an Unable-To-Grow-Stack exception occurs. The thread is allowed to continue execution, but must be aware that it has at most 4KB of stack remaining.

Obtaining Information about a Page Range

DosQueryMem is used to obtain information about a range of pages within the virtual address space of the current process.

Each page in the virtual address space of a process is either free, private, or shared.

Each page within a memory object can be in one of two states, either committed or uncommitted.

A committed page has its access controlled by an access protection attribute. These protection attributes are read protection (PAG_READ), write protection (PAG_WRITE), execute protection (PAG_EXECUTE), and guard page protection (PAG_GUARD).

Protection Violations

OS/2 fully utilizes the memory protection capabilities of the 80386 microprocessor. OS/2 grants an application access to a memory object only

if the object has been explicitly allocated by the application or made available for use by the application.

If an application attempts to access memory that it is not assigned, the system interrupts the application and executes the exception handling routine for protection violations. Protection violations can be handled by the application (in its own exception handling routines) or by OS/2. If the protection violation is handled by the operating system, the system exception handling routine determines the cause of the exception, displays an error message, and then terminates the application.

It is usually not possible for an application to recover from a protection violation. Therefore, programmers should ensure that all pointers are valid. Because protection violations commonly occur during application debugging, each message displayed for a protection violation includes the contents of the registers when the violation occurred. If the violation occurred as a result of passing an invalid pointer to a memory function, an error code is returned by the memory function.

In earlier versions of OS/2, protection violations could be used to find bugs when an application accessed memory that was not allocated to the application. This approach will no longer work because memory objects can be larger than the size requested by the application because the memory objects are allocated on 4KB page boundaries. For example, a pointer to the 513th byte of a 512 byte memory object is valid and does not cause a protection violation. This means that programmers cannot always rely on protection violations to spot memory addressing errors.

Memory Suballocation and Using Heaps

There are times when a process requires only small amounts of memory rather than an entire memory object. It would be wasteful to allocate an entire page of memory when only a few bytes are necessary, so a mechanism is provided for applications to allocate a large block of memory and then suballocate portions of the memory as necessary to fulfill small requests from an application. This is done by creating a heap.

A *heap* is a region of storage within a memory object from which an application can allocate blocks of memory. A *memory block* is a piece of memory within a heap. The size of the memory block is rounded up to the next higher multiple of 8 bytes.

Because OS/2 allocates a 4KB page for each memory allocation, using a heap to suballocate amounts of memory smaller than 4KB is more efficient than using DosAllocMem.

When an application creates a heap, it can have OS/2 track the committing and decommitting of memory within the heap. When the application commits and decommits memory itself, it has to keep track of the access state of each page as they are accessed.

Applications use DosSubSetMem to initialize a memory object for suballocation, then use DosSubAllocMem and DosSubFreeMem to allocate and free the memory.

Memory is still committed in pages when an application uses suballocation. If the application suballocates 512 bytes, 4096 bytes will be committed. Accessing the 513th byte will not cause a protection violation, but you could be accessing memory that was suballocated by another thread in the process.

Shared Memory

Shared memory is memory that two or more applications can read from and write to. Shared memory is prepared in such a way that any application can receive a pointer to the memory and access the data. Applications must explicitly request access to shared memory; the shared memory is protected from applications that are not granted access.

There are two kinds of shared memory: named and unnamed. For named shared memory, any application that knows the name of the shared memory can access it. For unnamed shared memory, a pointer to the shared memory must be passed from the process that created the shared memory to the process being given access. Access can be granted to any application; it is not necessary that the process being granted access be related (parent-child) to the application that created the shared memory.

Since memory sharing is done by sharing linear addresses, the linear address range of the shared memory object is reserved in all process address spaces.

There are two basic methods of managing shared memory:

- In one method, two or more applications share the same memory at the same time. These applications read from and write to the memory object, usually controlling access to the memory by using a semaphore.
- In the other method of managing shared memory, one application prepares data in memory, then passes that memory to another application for further processing. The first application releases the memory after passing it along, so that only one application accesses the memory at a time.

Thread Local Memory

Thread local memory is a 128-byte region of thread-specific memory that OS/2 assigns to a thread when the thread is created. Thread local memory for each thread is always at the same virtual address, but each thread's local memory region is backed by different physical memory, so the memory is specific to each thread. This memory region is divided into 32 DWORDS (4 bytes per DWORD), and is normally used to store a small number of pointers.

A thread can allocate thread local memory by calling DosAllocThreadLocalMemory. Thread local memory is freed by calling DosFreeThreadLocalMemory.

Note that thread local memory should be used sparingly. You should not use the entire 128 byte region.

Using Memory Management

This section describes how to use the OS/2 memory management functions and configuration commands to control the use of memory for OS/2 applications.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Allocating Private Memory

An application can allocate regions of storage within the virtual address space of the process. Such an allocated region is called a memory object.

DosAllocMem is used to allocate a memory object. You specify a variable to receive the pointer that will address the new object, the amount of memory needed, and the allocation attributes and access protection attributes of the new memory object. When choosing the size of the memory object to allocate, remember that the maximum size of the memory object is defined when it is allocated, and memory objects cannot be resized.

When applications call DosAllocMem, the operating system reserves a range of private pages large enough to fulfill the specified allocation request from the private virtual address space of the subject process.

DosAllocMem will reserve this linear space and return zero if the allocation was successful. If it was unsuccessful, DosAllocMem will return an error code. An application should always test the return value before attempting to use the memory.

The following code fragment requests an allocation of 512 bytes. Remember that 4096 bytes will actually be allocated for this request:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

PBYTE   pb;
APIRET  ulrc;

ulrc = DosAllocMem((PVOID *) &pb,
                   512,
                   fALLOC); /* pb receives the base address of the */
                            /* 4KB memory object                   */

if (!ulrc) {           /* If the allocation was successful, ulrc == 0 */
    *pb = 3000;        /* Use the allocated memory                    */
}
```

In this example, DosAllocMem returns a 32-bit pointer to a 4096 byte committed memory object and allows the application to write to and read from the memory. This pointer is valid only for the 4096 bytes allocated by the system. An attempt to use the pointer outside the allocated memory will cause a protection violation.

Committing and Decommitting Page Ranges

If an application allocates a sparse memory object, no physical memory location is committed for the object. Memory in a sparse object must be committed before it can be used. DosSetMem is used to commit or decommit a range of previously allocated pages in a private or shared memory object. Applications can make specific address ranges within a memory object valid or invalid. Commitment and decommitment always take place in multiples of one or more pages.

Applications can also use DosSetMem to change the access protection attributes of a range of pages within a memory object.

The following code fragment requests allocation of 2MB of uncommitted memory and then commits 4096 bytes of the memory:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

APIRET  ulrc;
PBYTE   pb;

/* Allocate 16KB object */
ulrc = DosAllocMem((PVOID *) &pb,
                   2097152,
                   PAG_READ |
                   PAG_WRITE);

/* Commit 4KB           */
ulrc = DosSetMem(pb,
                 4096,
                 PAG_COMMIT |
                 PAG_DEFAULT);
```

An application can also allocate a large committed object and then decommit portions of it as they are no longer needed. Decommitment, like commitment, is done on page boundaries; an application can decommit no less than a 4096 byte page.

The following code fragment allocates 16384 bytes of committed memory and then decommits the first 4096 bytes of the memory:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

APIRET  ulrc;
PBYTE   pb;

ulrc = DosAllocMem((PVOID *) &pb,
                   16384,
                   fALLOC);    /* Allocate 16 K object */

ulrc = DosSetMem(pb,
                 4096,
                 PAG_DECOMMIT);            /* Decommit 4KB         */
```

After memory is decommitted, an attempt to access the decommitted memory will cause a protection violation.

You cannot pass an argument that crosses multiple memory objects. The function will return an error.

Establishing Access Protection

When an OS/2 application commits a memory object, it specifies the types of access permitted for the memory object. This can be done at the same time the memory object is allocated, with DosAllocMem, or at a later time, using DosSetMem.

Any combination of read, write, execute, or guard-page access can be set, but at least read or write access must be specified when the memory object is committed; it is not possible to commit an object with no access protection attributes.

The application can also use DosSetMem to change the access permission of pages within a previously committed memory object. An application can permit read access to one page of the memory object and write access to the rest.

When using DosSetMem, all the pages in the range being changed must be either committed or decommitted.

The following code fragment commits a region of two pages within a previously allocated memory object, and sets read-only access rights for the region.

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>
#include <stdio.h>

PVOID   pBaseAddress;       /* Pointer to the range of pages to be changed */
ULONG   ulRegionSize;       /* Size, in bytes, of the region to be changed */
ULONG   ulAttributeFlags;   /* Flag describing the page range              */
APIRET  ulrc;               /* Return code                                 */

ulRegionSize = 8192;        /* Specify a two-page region                   */

/* Obtain the base address  */
ulrc = DosAllocMem((PVOID *) &pBaseAddress,
                   ulRegionSize,
                   PAG_WRITE);

ulAttributeFlags = PAG_COMMIT |
                   PAG_READ;

ulrc = DosSetMem(pBaseAddress,
                 ulRegionSize,
                 ulAttributeFlags);

if (ulrc != 0) {
    printf("DosSetMem error: return code = %ld", ulrc);
    return;
}
```

Querying Memory Object Information

DosQueryMem is used to determine the allocation state and access protection for a specified memory object. The application can query an entire memory object or a range of pages within an object.

The following code fragment uses DosQueryMem to ensure that memory is committed before the application attempts to use the memory:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values  */
#include <os2.h>
#define HF_STDOUT 1       /* Standard output handle */

PBYTE   pb;         /* Base address of an allocated object */
ULONG   ulSize,
        ulFlags,
        ulWritten;
APIRET  ulrc;       /* Return Code                         */

ulSize = 4096;

ulrc = DosAllocMem((PVOID *)&pb,
                   16384,
                   PAG_COMMIT |
                   PAG_WRITE);

ulrc = DosQueryMem(pb,
                   &ulSize,
                   &ulFlags);     /* Queries first 4096 bytes */

if (ulFlags & PAG_COMMIT) {     /* If memory is committed, use it */
    ulrc = DosWrite(HF_STDOUT,
                    "\r\n 4KB is committed.\r\n",
                    21,
                    &ulWritten);
}
```

Freeing Memory

When memory object is no longer needed, the application uses the DosFreeMem function to release the memory.

If applications do not release memory, the operating system either swaps the memory to the hard disk (if swapping is enabled) or uses memory that could be used by other applications. If OS/2 swaps the memory to the hard disk, the SWAPPER.DAT swap file could become very large. Therefore, applications should release memory as soon as it is no longer needed. Any memory that is still allocated when the application ends is released by OS/2.

DosFreeMem frees a private or shared memory object from the virtual address space of the process. The released pages are returned to the system.

The following code fragment allocates 8192 bytes of committed memory and then releases the memory:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

PBYTE   pb;
APIRET  ulrc;

ulrc = DosAllocMem((PVOID *) &pb,
                   8192,
                   fALLOC);    /* Allocate 8KB object */
    .
    .
    .
ulrc = DosFreeMem(pb);         /* Free the object     */
```

Using Suballocation and Heaps

This section describes how you can use DosAllocMem, DosSubSetMem, DosSubAllocMem, DosSubFreeMem, and DosSubUnsetMem to manage a memory heap.

Suballocating Memory

DosAllocMem can be used to create a memory heap.

Before an application can allocate small portions of the heap, it must use the DosSubSetMem function to set up the memory for suballocation. The size of the heap is rounded up to the next higher multiple of 8 bytes.

Then, the application uses DosSubAllocMem to allocate sections of the heap and the DosSubFreeMem function to release the memory.

DosSubAllocMem returns a 32-bit pointer to a block of memory. The pointer can be used to access the memory without further modification.

The following code fragment sets up 8192 bytes for suballocation and then allocates two small blocks of memory:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

APIRET  ulrc;
PBYTE   pbBase,
        pb1,
```

```
        pb2;

ulrc = DosAllocMem((PVOID *) &pbBase,
                   8192,
                   fALLOC);     /* Allocate 8K object    */

ulrc = DosSubSetMem(pbBase,
                    DOSSUB_INIT,
                    8192);      /* Set up object         */
                                /* for suballocation     */

ulrc = DosSubAllocMem(pbBase,
                      (PVOID *) &pb1,
                      100);     /* Suballocate 100 bytes */

ulrc = DosSubAllocMem(pbBase,
                      (PVOID *) &pb2,
                      500);     /* Suballocate 500 bytes */

ulrc = DosSubFreeMem(pbBase,
                     pb1,
                     100);      /* Free 1st suballocation*/

ulrc = DosSubAllocMem(pbBase,
                      (PVOID *) &pb1,
                      50);      /* Suballocate 50 bytes  */
```

Increasing the Size of a Heap

DosSubSetMem can also be used to increase the size of a previously initialized heap. The heap size can be increased up to the size of the memory object that contains it.

The size of the heap is rounded up to the next higher multiple of 8 bytes.

The following code fragment increases the size of a heap. Assume that the *Offset* variable was previously loaded with the virtual address of the memory object.

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>
#include <stdio.h>

PVOID   pOffset;    /* Address of the heap to be used for suballocation */
ULONG   ulFlags;    /* Flags describing the memory object being resized */
ULONG   ulSize;     /* Size in bytes to increase the size of the heap   */
APIRET  ulrc;       /* Return code                                      */

ulSize = 20000;     /* Indicate a heap size increase of 20000 bytes     */

/* DOSSUB_INIT set to initialize memory for suballocation */
ulFlags = DOSSUB_INIT |
          DOSSUB_SPARSE_OBJ;

ulrc = DosSubSetMem(pOffset,
                    ulFlags,
                    ulSize);

if (ulrc != 0) {
    printf("DosSubSetMem error: return code = %ld", ulrc);
    return;
}
```

In this example, the heap is incremented, and that memory commitment is managed internally within subsequent DosSubAllocMem calls.

When using DosSubSetMem to increase the size of the heap, the *Flags* parameter must have the same setting as when the heap was initialized.

Note: Do not call DosSetMem to change the allocation attribute or access protection attributes of any pages spanned by a memory object that the suballocation functions are managing. Otherwise, unpredictable results could occur.

Call DosSubUnsetMem when finished with the heap that was set up with DosSubSetMem. This enables the suballocation function to free the resources that it uses to manage the heap. When you are through with the memory object that the heap was part of, use DosFreeMem to free the memory object.

Allocating a Block of Memory from a Heap

DosSubAllocMem allocates a block of memory from a heap that was previously initialized by DosSubSetMem. This is used when an application needs an area of memory that is smaller than an entire heap.

The size of the memory block is rounded up to the next higher multiple of 8 bytes.

The following code fragment allocates a block of memory from a heap that was previously initialized by DosSubSetMem. Assume that the *Offset* variable has been set to the address of the initialized heap already.

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>
#include <stdio.h>

PVOID    pOffset;         /* The heap to suballocate from               */
PPVOID   ppBlockOffset;   /* Pointer to the variable where the offset of */
                          /* the suballocated memory block is returned   */
ULONG    ulSize;          /* Size in bytes of the memory block requested */
APIRET   ulrc;            /* Return code                                 */

ulSize = 102;             /* Ask for 102 bytes.  This will be rounded    */
                          /* to 104 bytes (a multiple of 8 bytes).       */

ulrc = DosSubAllocMem(pOffset,
                      &ppBlockOffset,
                      ulSize);

if (ulrc != 0) {
    printf("DosSubAllocMem error: return code = %ld",
           ulrc);
    return;
}
```

In this example, the address of the allocated block (from the heap) is stored in the *BlockOffset* variable.

Remember to call DosSubFreeMem to free this block of memory when you are finished with it.

Freeing Memory Blocks

DosSubFreeMem frees a block of memory that was previously allocated by DosSubAllocMem.

Call DosSubFreeMem to free a block of memory in the heap when you are finished with that memory block.

The following code fragment frees a block of memory that was previously allocated from a heap. DosSubFreeMem returns the block to the heap. Assume that the *Offset* variable has been previously set to the address of the initialized heap, and that the *BlockOffset* variable has been previously set to the address of the block to be returned to the heap.

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>
#include <stdio.h>

PVOID   pOffset;         /* Offset of the heap to which the          */
                         /* block is to be freed                     */
PVOID   pBlockOffset;    /* Offset of memory block to be freed       */
ULONG   ulSize;          /* Size in bytes of block to be freed       */
APIRET  ulrc;            /* Return code                              */

ulSize = 102;            /* Return 102 bytes.  This will be rounded  */
```

```
                          /* to 104 bytes (a multiple of 8 bytes).    */

    ulrc = DosSubFreeMem(pOffset,
                         pBlockOffset,
                         ulSize);

    if (ulrc != 0) {
        printf("DosSubFreeMem error: return code = %ld",
               ulrc);
        return;
    }
```

Ending the Use of the Heap

DosSubUnsetMem terminates the use of a heap within a memory object. All calls to DosSubSetMem must eventually be followed by a call to DosSubUnsetMem. This enables the suballocation function to free the resources that it uses to manage the heap.

The application must call DosSubUnsetMem before it frees the memory object that contains this heap (with DosFreeMem).

The following code fragment shows the termination of a heap. Assume that the address of the heap was placed into *Offset* already.

```
    #define  INCL_DOSMEMMGR   /* Memory Manager values */
    #include <os2.h>
    #include <stdio.h>

    PVOID   pOffset;    /* Offset of the heap whose use is being terminated */
    APIRET  ulrc;       /* Return code                                      */

    ulrc = DosSubUnsetMem(pOffset);

    if (ulrc != 0) {
        printf("DosSubUnsetMem error: return code = %ld",
               ulrc);
        return;
    }
```

Allocating and Freeing Thread Local Memory

Thread local memory is a small region of thread-specific memory assigned to a thread when it is started.

A thread can allocate thread local memory by calling DosAllocThreadLocalMemory. Thread local memory is freed by calling DosFreeThreadLocalMemory.

The thread local memory region is 32 DWORDS (4 bytes per DWORD) in size, and up to 8 DWORDS can be allocated with each call to DosAllocThreadLocalMemory. To allocate more than 8 DWORDS, a thread must call DosAllocThreadLocalMemory more than once.

Note that thread local memory should be used sparingly. You should not use the entire 128 byte region.

The following code fragment allocates six DWORDS in thread local memory and then frees these six DWORDS.

```
#define  INCL_DOSPROCESS
#define  INCL_DOSERRORS
#include <os2.h>
#include <stdio.h>          /* Needed for printf */

PULONG   pulMemBlock  = NULL;       /* Pointer to thread DWORDs returned */
APIRET   ulrc         = NO_ERROR;   /* Return code                       */

  ulrc = DosAllocThreadLocalMemory(6,
                                   &pulMemBlock );   /* Allocate 6 DWORDs */
```

```
if (ulrc != NO_ERROR) {
  printf("DosAllocThreadLocalMemory error: return code = %u\n",
         ulrc);
  return 1;
}

.
.
/* ... Use the thread-local memory block ... */
.
.

ulrc = DosFreeThreadLocalMemory(pulMemBlock);       /* Free the memory block */

if (ulrc != NO_ERROR) {
  printf("DosFreeThreadLocalMemory error: return code = %u\n",
         ulrc);
  return 1;
}
```

Using Shared Memory

This section describes how you can use DosAllocSharedMem, DosGiveSharedMem, DosGetSharedMem, and DosGetNamedSharedMem to use shared memory.

There are three types of shared memory:

- Named, where a process allocates and names shared memory. A second process can use the shared memory by using DosGetNamedSharedMem, specifying the name of the shared memory.
- Unnamed giveable, where a process gives another process access to shared memory with DosGiveSharedMem, using the PID of the second process.
- Unnamed gettable, where a process knows the address of shared memory they want to use (allocated by another process), gets permission to use that memory with DosGetSharedMem, and then uses the shared memory.

The difference among these different types of shared memory is not in how the memory is allocated by the original process, but in how the second process gets permission to use and access to the shared memory. All three types of shared memory are allocated with the same call: DosAllocSharedMem, using various parameters.

Using Named Shared Memory

An application uses DosAllocSharedMem to allocate shared memory. When allocating the shared memory, an application can assign a unique name to the memory. Any application that has the name of the shared memory can use DosGetNamedSharedMem to retrieve a pointer to the memory. This makes it possible for two or more applications to share memory at the same time.

The name of a shared memory object has the following form:

```
\sharemem\name
```

The "\sharemem\" is required. The "name" parameter can be any name that conforms to the rules for an OS/2 file name. No file is actually created for the memory object. There is no actual "\sharemem\" subdirectory.

The following code fragment allocates 65536 bytes of named shared memory with the name "\sharemem\mymem".

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

APIRET  ulrc;
CHAR    szMem[] = { "\\sharemem\\mymem" };
PULONG  pulPb;

ulrc = DosAllocSharedMem((PVOID *) &pulPb,
```

```
                          szMem,
                          65536,
                          fALLOC);

    *pulPb = 2762;
```

Once the named memory is allocated, any other process can retrieve a pointer to the named memory by using DosGetNamedSharedMem.

The following code fragment retrieves a pointer to the named memory allocated above:

```
    #define  INCL_DOSMEMMGR   /* Memory Manager values */
    #include <os2.h>
    #define HF_STDOUT 1       /* Standard output handle */

    APIRET  ulrc;
    CHAR    szMem[] = { "\\sharemem\\mymem" };
    PULONG  pulPb2;
    ULONG   ulWritten;

    ulrc = DosGetNamedSharedMem((PVOID *) &pulPb2,
                                szMem,
                                PAG_READ |
                                PAG_WRITE);

    if (*pulPb2 == 2762)
        ulrc = DosWrite(HF_STDOUT, "\r\n Success!\r\n", 13,
                        &ulWritten);
```

Using Unnamed Giveable Shared Memory

A process allocates unnamed giveable shared memory by using DosAllocSharedMem with the object name (pszName parameter) set to NULL and the memory options (flag parameter) set to OBJ_GIVEABLE. The process allocating the memory must pass a pointer to the shared memory to another process. This is typically done by using some form of interprocess communication, such as a queue or a named pipe.

If a process allocates shared memory with the OBJ_GIVEABLE option, this process can validate the pointer in another process with DosGiveSharedMem.

The following code fragment allocates 24576 bytes (24KB) of unnamed giveable shared memory:

```
    #define  INCL_DOSMEMMGR   /* Memory Manager values */
    #include <os2.h>

    APIRET  ulrc;
    PBYTE   pb;

    ulrc = DosAllocSharedMem((PVOID *) &pb,
                             (PSZ) NULL,
                             24576,        /* amount of memory */
                             fALLOC |
                             OBJ_GIVEABLE); /* giveable memory */
```

Once the memory is allocated, the allocating process can pass the memory pointer to a second process through interprocess communication. The second process need not use DosGetSharedMem; the second process can just use the shared memory.

Using Unnamed Gettable Shared Memory

A process allocates unnamed gettable shared memory by using DosAllocSharedMem with the object name (pszName parameter) set to NULL and the memory options (flag parameter) set to OBJ_GETTABLE. The allocating process must pass a pointer to the shared memory to another process. This is typically done by using some form of interprocess communication, such as a queue or a named pipe.

If an application allocates shared memory with the OBJ_GETTABLE option, it passes a pointer to the shared memory to the second process. The second process gets access to the shared memory by using DosGetSharedMem to validate the passed pointer.

The following code fragment allocates 24576 bytes (24KB) of unnamed gettable shared memory:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

APIRET  ulrc;
PBYTE   pb;

ulrc = DosAllocSharedMem((PVOID *) &pb,
                         (PSZ) NULL,
                         24576,
                         fALLOC |
                         OBJ_GETTABLE); /* gettable memory */
```

Once the memory is allocated, the process can pass the memory pointer to a second process through interprocess communication. Once the second process receives the pointer, it validates the memory with DosGetSharedMem, as shown in the following code:

```
#define  INCL_DOSMEMMGR   /* Memory Manager values */
#include <os2.h>

APIRET  ulrc;
PBYTE   pb2;

ulrc = DosGetSharedMem(pb2,  /* validating the memory allocated by */
                             /* the allocating process            */
                       PAG_READ |
                       PAG_WRITE);
```

Message Management

This chapter describes the use of message files to hold an application's text messages to the user. Keeping messages in a message file simplifies changing those messages, for example when maintaining versions of the same application for different countries.

The following topic is related to the information in this chapter:

- National language support

About Message Management

For full-screen applications, text messages-used by an application to display information to the user-can be held in a *message file* .

To keep and maintain messages in a message file, create a separate message file for each national language you intend to support. Use the MKMSGF utility program to create a binary version of a message file, and the MSGBIND utility program to bind the binary file to the application's executable file. Binding the messages to an application's executable file inserts the messages into the .EXE file. This enables faster access (but, of course, a larger .EXE file).

OS/2 provides several functions to assist in message management. Applications can get messages from a message file, substitute variable information into the messages, and write messages to the screen or to a file.

Code Page Considerations
You can have versions of the message file for each code page you choose to support. When you use MKMSGF to create the message files, the utility will insert the code page information and link together the message files of the different versions.

When message files are linked together in this manner, DosGetMessage will search for the appropriate version of the message for the code page that is active at the time the function is called. If there is no version of the message for the current code page, the function will return the first version of the message, no matter which code page it is associated with.

Note: To create portable source code for message files, make sure you specifically set the code page in CONFIG.SYS using the CODEPAGE statement. If you have the wrong code page, or use the default code page, which could be wrong, DosGetMessage returns error code 37 (ERROR_CODE_PAGE_MISMATCHED).

Using Message Management

DosGetMessage retrieves a message from the specified system message file, and inserts variable information into the body of the message.

DosInsertMessage inserts variable text-string information into the body of the message, but does not retrieve the message.

DosPutMessage sends the message in a buffer, usually to a display screen. DosPutMessage formats the screen buffer to prevent words from being split at the end of a line.

DosQueryMessageCP retrieves the message file list of code pages and language identifiers.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Message Retrieval and String Substitution

DosGetMessage retrieves a message from a system message file, and inserts variable text-string information into the message.

In the following code fragment, the message file is "D:\MESSAGE\AUTOMSG.MSG". The third message within the message file contains the string "%1 Error at Station %2". The application calls DosGetMessage to convert this message into the string "Automation Failure Error at Station 69B".

```
#define INCL_DOSMISC   /* Miscellaneous values */
#include <os2.h>
#include <stdio.h>

UCHAR   *ucIvTable[2];    /* Table of variables to insert       */
ULONG    ulIvCount;       /* Number of variables                */
UCHAR    ucDataArea[80];  /* Message buffer (returned)          */
ULONG    ulDataLength;    /* Length of buffer                   */
ULONG    ulMsgNumber;     /* Number of the message              */
UCHAR    ucFileName[40];  /* Message file path-name string      */
ULONG    ulMsgLength;     /* Length of message (returned)       */
UCHAR    ucField1[20];    /* String to substitute into variable */
                          /* field %1 of the message            */
UCHAR    ucField2[20];    /* String to substitute into variable */
                          /* field %2 of the message            */
APIRET   ulrc;            /* Return code                        */

strcpy(ucField1,
       "Automation Failure");   /* Define the field with which to  */
                                /* perform the first substitution  */

strcpy(ucField2,
       "69B");                  /* Define the field with which to           */
                                /* perform the second substitution          */

ucIvTable[0] = ucField1;       /* Set up the array of pointers to          */
ucIvTable[1] = ucField2;       /* substitute strings                       */

ulIvCount = 2;                 /* Two variable message fields in message   */

ulDataLength = 80;             /* Data buffer that will receive the        */
                               /* complete message is 80 bytes in size     */

ulMsgNumber = 3;               /* Specify the third message in the         */
                               /* message file                             */

strcpy(ucFileName,
       "D:\\MESSAGE\\AUTOMSG.MSG");
       /* Path name of the message file           */
```

```
ulrc = DosGetMessage(ucIvTable,
                     ulIvCount,
                     ucDataArea,
                     ulDataLength,
                     ulMsgNumber,
                     ucFileName,
                     &ulMsgLength);

if (ulrc != 0) {
    printf("DosGetMessage error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the *DataArea* buffer contains the complete message (with the two variable fields appropriately updated), and the *MsgLength* variable contains the length of the message that was placed into the *DataArea* buffer.

If an error occurs (that is, if the return code does not equal 0), a message that is related to the error will be placed in the message buffer. See the DosGetMessage API reference in *Control Program Programming Reference* for a description of the default messages that can be placed into the user's buffer if an error occurs during the processing of these requests.

Text String Substitution in Memory

DosInsertMessage inserts variable text-string information into a message that resides within program memory.

In the following code fragment, the message resides in a program string variable named *Message* . The message is the string "%1 Error at Station %2". The application calls DosInsertMessage to convert this message into the string "Automation Failure Error at Station 69B".

```
#define INCL_DOSMISC   /* Miscellaneous values */
#include <os2.h>
#include <stdio.h>

UCHAR   *ucIvTable[2];              /* Table of variables to insert         */
ULONG    ulIvCount;                 /* Number of variables                  */
UCHAR    ucMsgInput[40] = "%1 Error at Station %2";     /* The input message */
ULONG    ulMsgInLength;             /* Length of input message              */
UCHAR    ucDataArea[80];            /* Message buffer (returned)            */
ULONG    ulDataLength;              /* Length of updated message buffer     */
ULONG    ulMsgLength;               /* Length of updated message (returned) */
UCHAR    ucField1[20];              /* String to substitute into variable   */
                                    /* field %1 of the message              */
UCHAR    ucField2[20];              /* String to substitute into variable   */
                                    /* field %2 of the message              */
APIRET   ulrc;                      /* Return code                          */

strcpy(ucField1,
       "Automation Failure");       /* Define the field with which to  */
                                    /* perform the first substitution  */

strcpy(ucField2,
       "69B");                      /* Define the field with which to  */
                                    /* perform the second substitution */

ucIvTable[0] = ucField1;     /* Set up the array of pointers to */
ucIvTable[1] = ucField2;     /* substitute strings              */

ulIvCount = 2;                          /* Two variable message fields in  */
                                        /* message                         */

ulMsgInLength = strlen(ucMsgInput);     /* Length of input message         */

ulDataLength = 80;                      /* Data buffer that will receive   */
                                        /* the complete message is 80      */
                                        /* bytes in size                   */

ulrc = DosInsertMessage(ucIvTable,
                        ulIvCount,
                        ucMsgInput,
                        ulMsgInLength,
                        ucDataArea,
                        ulDataLength,
                        &ulMsgLength);
```

```
if (ulrc != 0) {
    printf("DosInsertMessage error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the *DataArea* buffer contains the complete message (with the two variable fields appropriately updated), and the *MsgLength* variable contains the length of the message that was placed into the *DataArea* buffer.

If an error occurs (that is, if the return code does not equal 0), a message that is related to the error will be placed in the message buffer. See the DosGetMessage API reference in *Control Program Programming Reference* for a description of the default messages that can be placed into the user's buffer if an error occurs during the processing of these requests.

Writing Messages

DosPutMessage is used to write the message to the screen or to a file on a disk.

The following code fragment writes the message string contained in *MessageBuffer* to the file specified by *FileHandle* . The message string has already been placed in *MessageBuffer* using either DosGetMessage or DosInsertMessage. *MsgLength* was set to the length of the message string by the same call that put the message into *MessageBuffer* .

```
#define INCL_DOSMISC   /* Miscellaneous values */
#include <os2.h>
#include <stdio.h>


HFILE   hfFileHandle;           /* Handle of output file or device */
ULONG   ulMsgLength;            /* Length of message buffer        */
UCHAR   ucMessageBuffer[80];    /* Message buffer                  */
APIRET  ulrc;                   /* Return code                     */

ulrc = DosPutMessage(hfFileHandle,
                     ulMsgLength,
                     ucMessageBuffer);

if (ulrc != 0) {
    printf("DosPutMessage error: return code = %ld",
           ulrc);
    return;
}
```

To write a message to the screen, use the standard output file handle.

Code Page Information Associated with Message Files

DosQueryMessageCP obtains a list of code page identifiers and language identifiers that are associated with a specified message file.

In the following code fragment code page and language identifiers associated with the message file "D:\MESSAGE\AUTOMSG.MSG" are retrieved.

```
#define INCL_DOSMISC   /* Miscellaneous values */
#include <os2.h>
#include <stdio.h>

UCHAR    ucBufferArea[20];  /* Buffer for the returned list  */
ULONG    ulBufferLength;    /* Length of the buffer area     */
UCHAR    ucFileName[40];    /* Message file path-name string */
ULONG    ulDataLength;      /* Length of the returned data   */
APIRET   ulrc;              /* Return code                   */

strcpy(ucFileName,
       "D:\\MESSAGE\\AUTOMSG.MSG");
```

```
       /* Path name of message file */

ulBufferLength = 20;                          /* Length of buffer area     */

ulrc = DosQueryMessageCp(ucBufferArea,
                         ulBufferLength,
                         ucFileName,
                         &ulDataLength);

if (ulrc != 0) {
    printf("DosQueryMessageCp error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the *BufferArea* buffer contains a set of information concerning the code page identifiers and language identifiers that are associated with the message file.

National Language Support

Many applications need to be independent of a particular language. Rather than being hard-coded in English, they want to support, for example, an English version of the application, and a French version, and a German version; preferably without having to change the program code for each version. Meeting this requirement is simplified through the use of such resources as string tables, menu templates, dialog templates, accelerator tables, and through the use of code pages.

This chapter describes the functions an application uses to be *NLS enabled*, or language independent.

The following topic is related to the information in this chapter:

- Message management

About National Language Support

The support of national languages by applications requires the following considerations:

- Displayed text must be translated into the appropriate language.
- Symbols or icons might not convey the same meaning in all countries. Alternative designs for different countries might be necessary.
- Translation changes the length of text strings.
- Different languages often have different text characters.

The use of national language resource files can help with the first three items, and the ability of the application to receive input and display output in any ASCII code page can help with the last item.

The use of ASCII code page 850 avoids many of the problems in this area, since it contains most of the characters required for Latin-1 languages, which include much of Western Europe and North and South America. However, older programs use code page 437 for U.S. English, and code pages 860, 863, and 865 for various languages. The code page applies to both input and output data.

Note: Code page 850 was used for translating Presentation Manager text. Use code page 850 whenever possible for all Presentation Manager applications that might require translation.

National Language Resource Files

When creating an application, define national language dependencies in resources that are held in resource files separate from the program

code. That is:

- Keep pure text strings in string tables.
- Keep menus in menu templates.
- Keep dialog boxes in dialog templates.
- Keep accelerators in accelerator tables.

The language displayed by the application can then be changed by translating the resources, in most cases without changing the application.

However, when translating from one language to another, the length of a text string can change substantially. For example, when translating from English to German, the length of a text string can double in length.

The following table furnishes a general idea of the amount of expansion that can be expected during translation.

Translation Expansion

For English Phrases	Translation Expansion Factors
Up to 10 characters	101 - 200%
11 - 20 characters	81 - 100%
21 - 30 characters	61 - 80%
31 - 50 characters	41 - 60%
51 - 70 characters	31 - 40%
Over 70 characters	30%

When designing your dialog boxes and text string messages, add white space to allow for the expansion that will occur when the text is translated. You might have to adapt the application program to allow for the change in the length of text strings after they are translated. For example, a change in the length of a text string can cause it to become misaligned with other displayed objects.

You can also use the Dialog Box Editor to adjust for misalignments, or to change the size of the dialog box. This would enable you to leave your application program unchanged.

Text strings explicitly displayed by the application program are more of a problem. You will have to include program code that can handle text strings of varying length and format them at runtime according to their size.

Language-Specific Versions of NLS-Enabled Applications

There are two methods of creating a specific national language version of a program designed to handle more than one national language. The choice of the method depends on the amount of available disk space and whether the user wants to change between different languages once the program is installed. The two methods are:

- Statically link the resources to the application's .EXE files. The executable files are then language-specific and cannot be changed to another national language. The specific .EXE files are then sent to the user.
- Place the resources into a language-specific, dynamic link library. Designate one library file for each national language. Selecting a particular library file for use with the application gives the desired version of the program. Using this method, all national languages can be shipped with the product; selection of the national language occurs during installation (for example, by naming a specific .DLL file). It is possible to change the national language setting while the program is operating.

About Code Page Management

A code page is a table that defines how the characters in a language or group of languages are encoded. A specific value is given to each character in the code page. For example, in code page 850 the letter "ñ" (lowercase) is encoded as hex A4 (decimal 164), and the letter "Ñ" (uppercase) is encoded as hex A5 (decimal 165).

Code page management enables a user to select a code page for keyboard input, and screen and printer output before starting an application, a system command, or a utility program in the OS/2 multitasking environment.

This means that a user in a particular country, such as England (code page 850), Norway (code page 865), or a language region such as Canadian French (code page 863) can use a code page that defines an ASCII-based character set containing characters used by that particular country or language.

Installable code page files include keyboard translate tables, display character sets, printer character sets, and country/language information for each code page supported.

Of particular interest are two code pages:

- Code Page 850
- Code Page 437

Code Page 850 (CP850)
Code Page 850 is also called the Latin-1, multilingual code page. This code page supports the alphabetic characters of the Latin-1-based languages. It contains characters required by 13 languages used in approximately 40 countries.

CP850 also provides the flexibility to develop new applications based on non-Latin-based or special industry-based code pages.

Code Page 850 supports countries using the following languages:

Belgian French	Canadian French
Danish	Dutch
Finnish	Flemish
French	German
Italian	Norwegian
Portuguese	Spanish
LAD Spanish	Swedish
Swiss French	Swiss German
U.K. English	U.S. English

Code Page 437 (CP437)
Code Page 437 is the standard personal computer code page.

The lower 128 characters are based on the 7-bit ASCII code. The upper 128 characters contain characters from several European languages (including part of the Greek alphabet) and various graphic characters. However, some of the accented characters, such as those used in the Nordic countries, are not represented. The missing characters are available in other code pages (code page 850 will usually contain the desired characters).

Note: Some of the 256 symbols that can be displayed are printer control characters, and are not printed.

ASCII and EBCDIC Code Page Support

The two leading character-coding systems are ASCII and EBCDIC. Presentation Manager applications can use an EBCDIC code page instead of an ASCII code page. Code pages based on both systems are supported by OS/2.

Any code page that either is defined in the CONFIG.SYS file, or is one of the EBCDIC code pages supported, can be selected.

Code Page Preparation

During system initialization, the code pages specified in the CODEPAGE statement are prepared to enable run-time code page switching of the display, the keyboard, the printer, and the country information. The display, keyboard, and printer must be defined in a DEVINFO statement in order to be prepared. Country information is prepared for the system country code specified in the COUNTRY statement.

If a resource cannot be prepared for the selected code page during system initialization, it is prepared for a default code page. The following are the defaults:

- A keyboard layout defaults to the code page of the translate table designated as the default layout in the KEYBOARD.DCP file. The default layout is based on the national code page of its associated country. You must explicitly specify KEYBOARD.DCP in the DEVINFO statement for the keyboard in CONFIG.SYS.
- The display defaults to the code page of ROM_0 for the device.

 (ROM_0 means a device default code page that is the device native code page or the lowest addressed ROM code page.)
- The printer defaults to the code page of ROM_0 for the device.

 (ROM_0 means a device default code page that is the device native code page or the lowest addressed ROM code page.)
- The country information defaults to the code page of the first entry found in the COUNTRY.SYS file for the country code. Each entry is the same information for a given country code, but is encoded in a different code page. The first entry is based on the preferred country code page.

If country information cannot be prepared at system initialization because it is not found in the COUNTRY.SYS file, for a code page selected with the CODEPAGE statement, then it is prepared (maintained for run-time code page switching in memory) in the default code page. Similarly, a keyboard layout is prepared in its default code page if it cannot be prepared in the selected code page, because it is not found in the KEYBOARD.DCP file.

COUNTRY.SYS contains one default entry per country code, and KEYBOARD.DCP contains one default entry per keyboard layout based on these assignments.

Code Page Functions

At the system level, OS/2 switches the code pages of supported displays and printers to agree with the code page of the process sending the output. At the application level, OS/2 functions enable a process to control code page assignments.

Using Code Pages

OS/2 provides applications with several functions to obtain information about and manipulate code pages. These functions enable applications to determine and set the current code page.

OS/2 code page management functions enable applications to read keyboard input and write display and printer output for multiple processes using ASCII-based data encoded in different code pages.

The system switches to the required code page, for a code-page-supported device, before input or output.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Querying Code Page Support and the Current Code Page

DosQueryCp is used to determine the code page of the current process and the prepared system code pages. The following code fragment shows how to get the current code page, and then up to three other prepared pages:

```
#define INCL_DOSNLS   /* National Language Support values */
#include <os2.h>

ULONG  ulCpList[8];
ULONG  ulCpSize;
APIRET ulrc;    /* Return code */
```

```
ulrc = DosQueryCp(sizeof(ulCpList),    /* Length of list          */
                  ulCpList,            /* List                    */
                  &ulCpSize);          /* Length of returned list */
```

The required code page is the current code page of the process at the time it opens a device, or a specific code page selected by the process with a set-code-page function. A character set can also be specified for some devices, for example, for some printers.

The country functions retrieve country- and language-dependent information in the current code page of the calling process, or in a code page selected by the process.

Setting the Code Page for Text Characters

Each process has a code page tag maintained by OS/2. A *code page tag* is the identifier of the current code page for the process.

A child process inherits the code page tag of its parent. The default code page for the first process in a session is the same as the session code page. The default code page for a new session is the primary code page specified in the CODEPAGE configuration statement.

To change the code page tag of a process, call DosSetProcessCp. This will not change the process code page tag of its parent or any child process.

Obtaining the Case Map String

DosMapCase performs case mapping on a string of binary values that represent ASCII characters.

The case map that is used is the one in the country file that corresponds to the system country code or selected country code, and to the process code page or selected code page. The default name of the country file is COUNTRY.SYS.

Obtaining the DBCS Environment Vector

DosQueryDBCSEnv obtains a double-byte character set (DBCS) environment vector that resides in the country file. The default name of the country file is COUNTRY.SYS.

The vector corresponds to the system country code or selected country code, and to the process code page or selected code page.

The following code fragment shows how to use DosQueryDBCSEnv:

```
#define INCL_DOSNLS   /* National Language Support values */
#include <os2.h>
#include <stdio.h>

ULONG         ulLength;             /* Length of data area provided         */
COUNTRYCODE   ccStructure;          /* Input data structure                 */
UCHAR         ucMemoryBuffer[12];   /* DBCS environmental vector (returned) */
APIRET        ulrc;                 /* Return code                          */

ulLength = 12;                      /* A length of 12 bytes is sufficient   */
                                    /* to contain the DBCS data returned    */

ccStructure.country = 0;            /* Use the default system country code  */

ccStructure.codepage = 0;           /* Return DBSC information for the      */
                                    /* caller's current process code page   */

ulrc = DosQueryDBCSEnv(ulLength,
                       &ccStructure,
                       ucMemoryBuffer);

if (ulrc != 0) {
    printf("DosQueryDBCSEnv error: return code = %ld",
```

```
            ulrc);
    return;
}
```

On successful return, the buffer *MemoryBuffer* will contain the country dependent information for the DBCS environmental vector.

Instead of the single-byte character set (SBCS) representation used for Latin text, some Asian countries use code pages that consist of double-byte character set characters, in which each character is represented by a two-byte code. The DBCS code pages enable single-byte data, double-byte data, or mixed (single-byte and double-byte) data.

Obtaining Formatting Information

DosQueryCtryInfo obtains country dependent formatting information that resides in the country file. The default name of the country file is COUNTRY.SYS.

The information corresponds to the system country code or selected country code, and to the process code page or selected code page.

Obtaining Collating Information for SORT

DosQueryCollate obtains a collating sequence table (for characters 00H through FFH) from the country file. The default name of the country file is COUNTRY.SYS. The SORT utility program uses this table to sort text according to the collating sequence.

The collating table returned corresponds to the system country code or selected country code, and to the process code page or selected code page.

Pipes

Communication between processes is valuable in a multitasking operating system to enable concurrent processes to work together. Pipes are one of three forms of interprocess communication (IPC), the other forms of IPC being semaphores and queues.

This chapter describes how to create, manage, and use pipes. Pipes enable two or more processes to communicate as if they were reading from and writing to a file.

The following topics are related to the information in this chapter:

- Memory (shared memory)
- Program execution and control
- Semaphores
- Queues

About Pipes

A *pipe* is a named or unnamed buffer used to pass data between processes. A process writes to or reads from a pipe as if the pipe were standard input or standard output. A parent process can use pipes to control the input that a child process receives and to receive the output that the child process produces.

There are two types of pipes-named and unnamed.

Unnamed Pipes

An unnamed pipe is a circular buffer in memory. The buffer has *in* and *out* pointers that are maintained by the system.

An unnamed pipe can transfer information only between related processes. A child process started by a parent process with DosExecPgm inherits the handles to any unnamed pipes created by its parent. This inheritance enables the parent process and the child process to use the unnamed pipe to communicate with one another. This type of pipe is typically used to redirect the standard input and standard output of a child process.

To do this, a process opens a pipe and duplicates the read and write handles of the pipe as the standard input and standard output files for the child process. Once the handles are duplicated, the parent process can use DosExecPgm to start the child process. When the child process reads and writes to its standard input and standard output handles, it is reading and writing to the pipe. The parent process can also communicate with the child process through the pipe.

Using an unnamed pipe, a text editor could run another program, such as a compiler or assembler, and display the output of the compiler or assembler within the editor.

DosCreatePipe creates an unnamed pipe. This function returns two file handles for the pipe, one for writing to the pipe and another for reading from the pipe. A process can then write to the pipe by using DosWrite and read from the pipe by using DosRead.

A pipe exists until both handles are closed. The order in which the handles are closed is sometimes important. For example, DosWrite might wait for data to be read from the pipe before completing its operation. In this case, the read handle is closed before the write handle is closed, writing to the pipe generates an error.

No control or permission mechanisms or checks are performed on operations to unnamed pipes.

Named Pipes

Named pipes enable related or unrelated processes on either the same computer system or different systems to communicate with each other. Any process that knows the name of a pipe can open and use a named pipe. In addition, named pipe data can be transparently redirected across a network, such as a local area network (LAN). (Unnamed pipes, by contrast, can be used only by related processes that are on the same computer system.)

One process (the *server process*) creates the pipe and connects to one end of it. Other processes that access the named pipe are called *client processes* ; they connect to the other end of the pipe. The server and client processes can then pass data back and forth by reading from and writing to the pipe. The server process controls access to the named pipe.

The client process can be either local or remote. A local client process is one that runs on the same computer system as the server process. A remote client process runs on a different system and communicates with the server process across a local area network (LAN).

When the server process creates a named pipe with DosCreateNPipe, it must specify the direction that data will flow through the pipe. The process specifies an inbound pipe if it intends to read data from the client process, an outbound pipe if it intends to write data to the client process, or a duplex pipe if it intends to read from and write to the client process.

The server process also specifies whether data passes through the pipe as bytes or messages. A message is a block of data, with a system-supplied header, that is read or written as a single unit. The server and client processes define the size and format of a message.

The server process also specifies whether child processes will inherit the named pipe and how information will be read from and written to the pipe. If the server specifies wait mode, DosRead will be blocked (it will not return to the process) until data is available in the pipe, and DosWrite will be blocked until there is enough room in the pipe to contain the entire data buffer. If the server specifies no-wait mode, reading from an empty pipe or writing to a full pipe immediately returns an error value.

A named pipe consists of two pipe buffers, one for each direction of communication. However, each end of the pipe has only one handle associated with it. The server receives the handle for its end when it creates the pipe with DosCreateNPipe. The client receives the handle for its end when it opens the pipe with DosOpen.

The server and the client use their respective handles both to read from the pipe and to write to it. (This is in contrast to unnamed pipes, for which both the server and the client read from one handle and write to another.) In other words, data that is written by the process at one end of the pipe is read by the process at the other end.

A named pipe can have multiple instances, up to the number specified when the pipe is first created. Pipe instances are actually separate pipes-that is, unique sets of pipe buffers with unique handles-that share the same name. The ability to create multiple pipe instances enables the server to communicate with multiple client processes at the same time.

Server-Client Communications Using Named Pipes

A server process initiates a connection to a client process by using DosConnectNPipe. Once the pipe has been connected by the server

process, the client process must open the pipe by using DosOpen to complete the connection. If no client process has opened the pipe when the server process calls DosConnectNPipe, the function either waits until a client opens the pipe or returns ERROR_PIPE_NOT_CONNECTED, depending on whether the server process created the pipe to wait for data.

Each client process requires a separate instance of a pipe. For example, if a server process creates a named pipe and specifies that four instances of the pipe can be created, the process can then create three more instances of the named pipe (for a total of four pipes, including the original). Each instance has the same name, but each has a unique pipe handle. The process can then connect each pipe to the server. (Each instance must be connected by an explicit use of DosConnectNPipe.) In this example, the server must use each function four times to create and connect four separate instances of the named pipe. Each pipe is then available to a separate client process.

If a client process receives the ERROR_PIPE_BUSY return value from DosOpen, no instances of the given pipe are available. The process can use DosWaitNPipe to wait for an instance to become available. The function waits until an instance is free or until the specified time interval elapses. When an instance becomes free, the process can open the pipe. If several processes are waiting for an instance to become available, the system gives the named pipe to the process that has been waiting the longest.

The server process can disconnect a client process from a pipe by using DosDisConnectNPipe. Ideally, the client process closes the pipe by using DosClose before the server process disconnects the pipe. If the client process has not closed the pipe when the server process disconnects it, the server process forces the pipe closed and the client process subsequently receives errors if it attempts to access the pipe. Note that forcing the closure of the pipe might discard data in the pipe before the client process has had an opportunity to read it.

A process can read and write bytes to a named pipe by using DosRead and DosWrite. A process can read or write messages by using DosTransactNPipe. Depending on the access mode, DosTransactNPipe writes a message to the pipe, reads a message from the pipe, or both. If a named pipe contains unread data or is not a message pipe, DosTransactNPipe fails.

Named pipes created with the NP_ACCESS_INBOUND or NP_ACCESS_OUTBOUND access mode cannot use the DosTransactNPipe function. If the named pipe's client uses the DosTransactNPipe function, the function returns error code 5 (ERROR_ACCESS_DENIED).

If it is reading from the pipe, DosTransactNPipe does not return until a complete message is read, even if the server process specified no-wait mode when the pipe was created.

A process can also read data from a named pipe without removing the data from the pipe by using DosPeekNPipe. This function copies the specified number of bytes from the pipe and returns the number of bytes of data left in the pipe and the number of bytes left in the current message, if any.

DosPeekNPipe also returns the state of the pipe. A named pipe can be in one of the following states:

Named Pipe States

State	Description
Connected	The pipe has been created and connected by the server process and has been opened by a client process. Only connected pipes can be written to or read from.
Closing	The pipe has been closed by the client process but has not yet been disconnected by the server process.
Disconnected	The pipe has been created by the server process but not connected, or has been explicitly disconnected and not yet reconnected. A disconnected pipe cannot accept a DosOpen request.
Listening	The pipe has been created and connected by the server process but has not yet been opened by a client process. A listening pipe is ready to accept a request to open. If the pipe is not in a listening state, DosOpen returns ERROR_PIPE_BUSY.

DosCallNPipe is used to open, write to, read from, and close a named message-format pipe.

Named pipes created with the NP_ACCESS_INBOUND or NP_ACCESS_OUTBOUND access mode cannot use the DosCallNPipe function. If the named pipe's client uses the DosCallNPipe function, the function returns error code 5 (ERROR_ACCESS_DENIED). This function is equivalent to calling DosOpen, DosTransactNPipe, and DosClose If no instances of the pipe are available, DosCallNPipe waits for an instance or returns without opening the pipe if the specified interval of time elapses.

A process can retrieve information about the handle state of a named pipe by using DosQueryNPHState. The handle state is a combination of the instance count, the access mode, and the pipe type (byte or message). DosQueryNPHState also specifies whether the process owning the handle is a server or client and whether the pipe waits if reading and writing cannot proceed.

A process can modify the handle state of a named pipe by using DosSetNPHState. For example, the process can change the reading mode for the pipe, enabling a process to read bytes from the pipe instead of messages.

Steps in Managing Server-Client Transactions

The following sequence summarizes the typical steps in the management of a named pipe:

1. The server process creates the pipe by calling DosCreateNPipe. DosCreateNPipe returns a handle for the server end of the pipe. (Note that the server uses the same handle to both read from and write to the pipe.) The pipe is now in the *disconnected state* and cannot be opened by a client process. The server process calls DosConnectNPipe to put the pipe into a *listening state* .

2. The pipe can now be opened by a client process.

3. A client process supplies the name of the pipe in a call to DosOpen and receives a pipe handle in return. (The client uses the same handle to both read from and write to the pipe.) The pipe is now in the *connected state* and can be read from or written to by the client.

4. The server and client processes communicate by calling DosRead and DosWrite. DosResetBuffer can be used to synchronize read and write dialogs. A server process that supports a large number of clients for a local named pipe can use DosSetNPipeSem and DosQueryNPipeSemState to coordinate access to the pipe. Server and client processes can also use DosTransactNPipe and DosCallNPipe to facilitate their communication.

5. After completing its transactions, the client process calls DosClose to close its end of the pipe. The pipe is now in the *closing state* and cannot be accessed by another client.

6. The server process calls DosDisConnectNPipe to acknowledge that the client has closed its end of the pipe. The pipe is now in the disconnected state again.

7. To enable another client process to open the pipe, the server must call DosConnectNPipe again. This puts the pipe back into the listening state. To end its access to the pipe, the server calls DosClose. When all of the handles for both ends of the pipe have been closed, the pipe is deallocated by the system.

Preparing a Named Pipe for a Client

A server process uses DosConnectNPipe to put a newly created named pipe into the listening state. The pipe must be in the listening state in order for a client process to gain access to the pipe by calling DosOpen.

After successfully opening the pipe and finishing its transactions, the client process calls DosClose to end its access to the pipe. The server process must acknowledge the close by calling DosDisConnectNPipe. It can then call DosConnectNPipe again to put the pipe into the listening state for the next client.

Together, DosConnectNPipe and DosDisConnectNPipe enable a server to create a named pipe and to reuse it for communication with different clients. Without these functions, the server would have to delete and re-create the pipe for each client.

Note: If multiple instances of a named pipe have been created, then each instance of the pipe must be put into the listening state before it can be opened by a client.

Facilitating Transaction Processing

DosTransactNPipe and DosCallNPipe facilitate the use of named pipes by combining other named pipe functions. Compared to calling the other functions separately, DosTransactNPipe and DosCallNPipe provide significant performance gains for applications that operate in a networked environment. They can also be used by local processes. However, both of these functions can be used only with duplex message pipes.

- DosTransactNPipe performs a transaction (a DosWrite followed by DosRead) on a duplex message pipe.

- DosCallNPipe has the combined effect of DosOpen, DosTransactNPipe, and DosClose, and is referred to as a *procedure call* . It provides an efficient means of implementing local and remote procedure call interfaces between processes.

Coordinating Access to a Local Named Pipe with Semaphores

When a process writes to a named pipe, the process at the other end (or handle) of the pipe might require notification that data is available to be read. Similarly, when a process reads from a named pipe, the process at the other end might require notification that write space has become available. As long as the communicating processes are on the same computer system, shared event semaphores and muxwait semaphores can be used to provide this notification. Using shared semaphores for this purpose is more efficient than dedicating a thread to periodically poll each pipe, particularly when a server process is communicating with a large number of client processes. Whenever data is available in the pipe, the system posts a semaphore to the server or client (whichever is reading from the pipe). This means that the reading process can use DosWaitEventSem or DosWaitMuxWaitSem to wait for data to arrive, rather than devote a thread to periodically polling the pipe.

A process associates a semaphore with a named pipe by using DosSetNPipeSem. First, create an event semaphore with DosCreateEventSem, specifying the initial state of the semaphore as *reset*. Then call DosSetNPipeSem to attach the event semaphore to a particular named-pipe handle. Up to two event semaphores can be attached to each named pipe, one for the server process and one for the client process. If there is already a semaphore associated with one end of the pipe, that semaphore is replaced. A process can check the state of the semaphores by using DosQueryNPipeSemState.

The server or client process must then call DosWaitEventSem. The particular thread that calls this function will block until data is either read from or written to the pipe. At that time, the system posts the event semaphore, enabling the blocked thread to resume its execution.

If a process requires notification whenever any one of multiple named pipes has been written to or read from, it can either attach the same event semaphore to multiple pipes, or it can create a muxwait semaphore:

- If the same event semaphore is attached to multiple pipes, then the *KeyHandle* parameter of DosSetNPipeSem is used to assign a unique value to each pipe. After the event semaphore has been posted, the process calls DosQueryNPipeSemState. This function returns information about each of the named pipes that are attached to the semaphore, including key-handle values. The calling process can use this information to determine which one of the named pipes has either data or write space available.
- To use a muxwait semaphore, a process first creates an event semaphore for each of the pipes that it wants to monitor. Each semaphore must then be attached to a pipe by calling DosSetNPipeSem. Again, a unique key-handle value must be assigned to each pipe.

 Next, the process calls DosCreateMuxWaitSem to create the muxwait semaphore, specifying DCMW_WAIT_ANY as one of the *flAttr* flags. The muxwait semaphore will consist of a linked list of the previously created event semaphores.

 The process calls DosWaitMuxWaitSem so that it will be notified the next time data is read from or written to any of the pipes. However, it must call DosQueryNPipeSemState to determine which one of the pipes is ready to be read from or written to.

Using Unnamed Pipes

Unnamed pipes are useful in applications that transfer data between related processes. They are commonly used to control the input and output of child processes by redirecting the standard input and output of the child process to a pipe controlled by the parent process.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Creating Unnamed Pipes

DosCreatePipe creates an unnamed pipe. Two handles are returned: one for read access to the pipe and one for write access. The pipe size specified is advisory; its actual size is dependent on the amount of available memory. If the size parameter is 0, the pipe is created with the default size, which is 512 bytes.

This example creates an unnamed pipe. The current process can use the unnamed pipe for communication between itself and a child process.

```
#define INCL_BASE    /* Queue values */
#include <os2.h>
```

```
#include <stdio.h>

HFILE    hfReadHandle;    /* Pointer to the read handle      */
HFILE    hfWriteHandle;   /* Pointer to the write handle     */
ULONG    ulPipeSize;      /* Pipe size                       */
APIRET   ulrc;            /* Return code                     */

ulPipeSize = 4096;        /* Ask for 4KB of internal storage */
                          /* for the pipe                    */

ulrc = DosCreatePipe(&hfReadHandle,
                     &hfWriteHandle,
                     ulPipeSize);

if (ulrc != 0) {
    printf("DosCreatePipe error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the *ReadHandle* variable contains the read handle for the pipe, and the *WriteHandle* variable contains the write handle for the pipe.

After a process creates a pipe, any child process started with DosExecPgm inherits the pipe handles. Using shared memory, the parent process can pass one of the pipe handles to the child process; thus, one process can store data in the pipe and the other can retrieve it.

Reading from and Writing to Unnamed Pipes

Applications use the OS/2 file system functions to read from and write to unnamed pipes.

The handles returned by DosCreatePipe are used as file handles to DosRead and DosWrite.

To write (or add data) to an unnamed pipe, call DosWrite, specifying the write handle of the pipe in DosWrite's file handle parameter. DosWrite requests to a pipe are processed in the order in which they are made. Multiple calls to DosWrite can be made before data is read (or removed) from the pipe. When the pipe becomes full, write requests are blocked until space is freed by read requests.

When the pipe is too full to hold the entire contents of the write request, the portion that will fit is written to the pipe before the writer is blocked.

To read from a pipe, call DosRead, specifying the read handle of the pipe in DosRead's file handle parameter. Subsequent calls to DosRead can empty the pipe if no further calls to DosWrite are made in the meantime.

Different threads writing to and reading from a pipe are not synchronized. It is possible for the pipe reader to obtain partial contents of a write request if the pipe becomes full during the write request.

If the process reading the pipe ends, the next DosWrite request for that pipe returns ERROR_BROKEN_PIPE.

Calling DosClose terminates access to an unnamed pipe. However, the pipe is not deleted from memory until all handles to the pipe have been closed, including any handles that were defined with DosDupHandle.

Redirecting Standard I/O for Child Processes

An application can use unnamed pipes to redirect the standard input and the standard output for a child process.

A typical use of an unnamed pipe is to read the output of a child process. An application creates a pipe and then duplicates the standard output handle. When the child process is started, its standard output will be written into the pipe, where the application can read and display it. The following code fragment shows how to do this:

```
#define INCL_DOSPROCESS
#define INCL_DOSNMPIPES

#include <os2.h>

#define PIPESIZE 256
#define HF_STDOUT 1      /* Standard output handle */
```

```
HPIPE hpR, hpW;
RESULTCODES resc;
ULONG ulRead, ulWritten;
CHAR achBuf[PIPESIZE],
     szFailName[CCHMAXPATH];

HFILE hfSave = -1,
      hfNew = HF_STDOUT;

DosDupHandle(HF_STDOUT,
             &hfSave); /* Saves standard output handle      */

DosCreatePipe(&hpR,
              &hpW,
              PIPESIZE); /* Creates pipe                      */

DosDupHandle(hpW,
             &hfNew); /* Duplicates standard output handle */


DosExecPgm(szFailName,
           sizeof(szFailName), /* Starts child process      */
           EXEC_ASYNC,
           (PSZ) NULL,
           (PSZ) NULL,
           &resc,
           "DUMMY.EXE");

DosClose(hpW);               /* Closes write handle to ensure     */
                             /* Notification at child termination */

DosDupHandle(hfSave,
             &hfNew);     /* Brings stdout back               */

/*
 * Read from the pipe and write to the screen
 * as long as there are bytes to read.
 *
 */

do {
    DosRead(hpR,
            achBuf,
            sizeof(achBuf),
            &ulRead);

    DosWrite(HF_STDOUT,
             achBuf,
             ulRead,
             &ulWritten);

} while(ulRead);
```

A parent process can also use unnamed pipes to communicate with a child process by redirecting both the standard input and the standard output for the child process. To do this, the parent process:

1. Uses DosDupHandle to redefine the read handle of one pipe as standard input (0000), and the write handle of the other pipe as standard output (0001).
2. Starts the child process with DosExecPgm.
3. Uses the remaining pipe handles to read and write to the pipes.

The parent process controls the meanings for standard I/O for the child process. Thus, when the child process uses standard I/O handles with DosRead and DosWrite, it reads from and writes to the pipes of its parent instead of reading from the keyboard and writing to the display.

Using Named Pipes

Named pipes are useful in applications that transfer data between processes. The processes using named pipes can be related, unrelated, or even on different computers.

Creating Named Pipes

The server process creates a named pipe by using DosCreateNPipe.

To create a named pipe, specify on the DosCreateNPipe function:

- The name of the pipe
- The access modes
- The type of pipe (byte or message)
- The sizes of the input and output buffers

DosCreateNPipe returns a pipe handle that can be used in subsequent pipe operations.

Each named pipe must have a unique name of the following form:

```
\PIPE\PipeName
```

The "\PIPE\" in the name above is required, but need not be uppercase. It is not the name of a subdirectory.

To open a pipe on a remote computer, the client process must specify the name of the server process that opened the pipe as part of the pipe name, as follows:

```
\\Server\PIPE\PipeName
```

"\\Server" in the name above is the name of the remote computer; again, "\PIPE\" is required.

The name parameter must conform to the rules for OS/2 file names, but no actual file is created for the pipe.

Named pipes created with certain access modes prevent the named pipes' clients from using certain functions. If the named pipe is created with access mode NP_ACCESS_INBOUND, the named pipe's client cannot use these functions:

- DosCallNPipe
- DosTransactNPipe
- DosPeekNPipe

If the named pipe is created with access mode NP_ACCESS_OUTBOUND, the named pipe's client cannot use these functions:

- DosCallNPipe
- DosTransactNPipe
- DosWrite

If the named pipe's client uses an incorrect function, the function returns error code 5 (ERROR_ACCESS_DENIED).

In the following code fragment, DosCreateNPipe creates a pipe named \pipe\pipe1 and supplies a unique handle identifying the pipe. *OpenMode* is set to NP_ACCESS_DUPLEX. This activates full duplex access to the named pipe. There will be no inheritance to child process, and no write-through (write-through only affects remote pipes). *PipeMode* is set to "NP_WMESG | NP_RMESG | 0x01". This specifies that the pipe should be read as a message stream for both reading and writing and an instance count of 1 (only one instance of the named pipe can be created at a time). The pipe will block on Read/Write if no data is available.

```
#define INCL_DOSNMPIPES   /* Named-pipe values */
#include <os2.h>
#include <stdio.h>

UCHAR   ucFileName[40];  /* Pipe name              */
HPIPE   hpPipeHandle;    /* Pipe handle (returned) */
ULONG   ulOpenMode;      /* Open-mode parameters   */
ULONG   ulPipeMode;      /* Pipe-mode parameters   */
ULONG   ulOutBufSize;    /* Size of the out-buffer */
ULONG   ulInBufSize;     /* Size of the in-buffer  */
ULONG   ulTimeOut;       /* Default value for      */
                         /* DosWaitNPipe time-out  */
                         /* parameter              */
APIRET  ulrc;            /* Return code            */

strcpy(ucFileName,
       "\\PIPE\\PIPE1");

ulOpenMode = NP_ACCESS_DUPLEX;           /* Full duplex, no inheritance,     */
                                         /* no write-through                 */

ulPipeMode = NP_WMESG |
             NP_RMESG |
             0x01;     /* Block on read and write, message                  */
                       /* stream, instance count of 1                       */

ulOutBufSize = 4096;   /* The outgoing buffer must be 4KB in size           */
```

```
ulInBufSize = 2048;    /* The incoming buffer must be 2KB in size           */

ulTimeOut = 10000;     /* Time-out is 10 seconds (units are in milliseconds) */

ulrc = DosCreateNPipe(ucFileName,
                      &hpPipeHandle,
                      ulOpenMode,
                      ulPipeMode,
                      ulOutBufSize,
                      ulInBufSize,
                      ulTimeOut);

if (ulrc != 0) {
    printf("DosCreateNPipe error: return code = %ld",
           ulrc);
    return;
}
```

Once the named pipe is created, the application can call DosConnectNPipe to connect a client process to the pipe.

Once a client process connects to the pipe, the process can read from and write to the pipe. The preceding example creates a byte pipe, so the process can use DosRead and DosWrite to read from and write to the pipe.

After the client process finishes using the pipe, the server process can disconnect the pipe by using DosDisConnectNPipe. The server process can either connect again or close the named pipe by using DosClose.

When a server process creates a named pipe, it defines the pipe to the system by specifying the file write-through mode, the inheritance mode, the access and blocking modes, the pipe type, the read mode, the size of the in and out buffers, and the instance count. The following list describes these modes, types, and buffers.

- The *file write-through mode* has significance only for communication with remote client processes. When the file write-through bit is set, data is sent across the network as soon as it is written; otherwise, OS/2 will in some cases hold data briefly in a local buffer before sending it across the network.
- The *inheritance mode* specifies whether or not the pipe handle will be inherited by a child process.
- The *access mode* specifies the direction in which data will flow through the pipe. The server creates an inbound pipe (a pipe with inbound access mode) if it intends to read data from the client process, an outbound pipe if it intends to write data to the client process, or a duplex pipe if it intends to both read from and write to the client process.
- The *blocking mode* specifies whether or not DosRead and DosWrite will block when no data is available to read, or there is no room to write data.
- The *pipe type* is the form in which a stream of data is written to the pipe. If the pipe is a byte pipe, the server and client processes write data as an undifferentiated stream of bytes. If the pipe is a message pipe, the processes write data as a stream of messages; messages are blocks of data, each with a system-supplied header, that are written as single units. The server and client processes define the size and format of a message.
- The *read mode* is the form in which data is read from the pipe. The data in a pipe that was created as a byte pipe can only be read as bytes; therefore, a byte pipe will always be in byte-read mode. The data in a message pipe, however, can be read either as messages or as bytes. (If it is to be read as bytes, DosRead skips over the message headers). Therefore, message pipes can be in either message-read mode or byte-read mode.

 Note: The terms "byte pipe" and "message pipe" always refer to the pipe type-the form in which data is written to the pipe. When the read mode of a pipe is being referred to, it is always explicitly identified as either message-read mode or byte-read mode.
- The *in* and *out buffers* can be up to 64KB in size. If the pipe will be read in message-read mode, and if the message size is known, the server can control how many messages the buffer will hold at one time by specifying the appropriate buffer size.
- The *instance count* is the maximum number of instances of the named pipe that can be created. A pipe instance is actually a separate pipe-that is, a unique set of pipe buffers with unique handles. However, the term "pipe instance" is used to distinguish pipes that share the same name from pipes with different names. Because a client process uses only the name of the pipe when opening it, the existence of multiple pipe instances is transparent to a client process.

Creating Multiple Instances of a Named Pipe
Although each named pipe must have a unique name, a server process can create multiple *instances* of a pipe, all of which have the same name. A pipe instance is actually a separate pipe-that is, a unique set of pipe buffers with unique handles.

The ability to create multiple pipe instances enables the server to communicate with multiple client processes at the same time. Because a client process uses only the name of the pipe when opening it, the existence of multiple pipe instances is transparent to a client process.

The *ICount* parameter of DosCreateNPipe specifies the maximum number of named pipe instances that can be created. (An unlimited number can also be specified.) This parameter is specified only when the first instance of a named pipe is created; any subsequent attempt to redefine the instance count will be ignored.

If the instance count is greater than 1, the server process can create additional pipe instances by specifying the same pipe name in subsequent calls to DosCreateNPipe. Generally, the attributes of the subsequent pipe instances are defined to be the same as those of the original pipe instance, because a client process that requests the pipe has no way of controlling which pipe instance will be assigned to it.

After an additional pipe instance has been created, it is used in the same manner as the original pipe instance. That is, the same sequence of named pipe functions is used in the control or management of all named pipe instances. (See Steps in Managing Server-Client Transactions for more information.)

Note: If all of the instances of a named pipe are in use when a client calls DosOpen, ERROR_PIPE_BUSY is returned. However, the client can wait for an instance of that pipe to become available by calling DosWaitNPipe.

Multiple instances of a named pipe can be created by different processes. That is, multiple server processes can create and use instances of the same named pipe to communicate with their clients.

Opening Named Pipes

A client process can open the client end of a named pipe by using DosOpen. DosOpen opens the client end of a pipe by name and returns a handle. The application must use the appropriate pipe name and access modes to open the pipe for reading, writing, or both. (To open a pipe on a remote computer, the client process must also specify the name of the computer system as part of the pipe name, as follows:

```
\\ComputerName\PIPE\PipeName.)
```

If a pipe name includes a remote LAN server name, DosOpen attempts to open the pipe on a remote computer. The server process can then read input from the client process through the pipe.

The following code fragment opens a remote pipe, reads from the standard input (usually the keyboard), and sends the information to the server process through the pipe:

```
#define INCL_DOSQUEUES   /* Queue values          */
#include <os2.h>

#define PIPESIZE 256
#define SERVER_PIPE_NAME "\\\\myserver\\pipe\\mypipe"
#define HF_STDIN 0       /* Standard input handle */

HPIPE   hp;
BYTE    abBuf[PIPESIZE];
ULONG   ulAction, ulRead, ulWritten;
APIRET  ulrc;

ulrc = DosOpen(SERVER_PIPE_NAME,
               &hp,
               &ulAction,
               0,
               FILE_NORMAL,
               FILE_OPEN,
               OPEN_ACCESS_READWRITE |
               OPEN_SHARE_DENYNONE,
               (PEAOP) NULL);

if (ulrc)
    DosExit(EXIT_PROCESS,
            0);                   /* Open pipe failed      */

do {                              /* Open pipe succeeded   */
    DosRead(HF_STDIN,
            abBuf,
            sizeof(abBuf),
            &ulRead);

    DosWrite(hp,
             abBuf,
             ulRead,
             &ulWritten);     /* Writes to the pipe    */
} while (ulRead > 2);         /* Stop on a blank line  */

DosClose(hp);
```

The client process checks the return value from DosOpen to verify that the pipe was actually opened. If the server process has not yet created the pipe, DosOpen returns an error. When the client process finishes using the pipe, it closes the pipe by using DosClose.

When a named pipe is opened, its initial state is set by the system to block read and write operations (blocking mode), and to read data as a byte stream (byte-read mode). However, the client can change these modes by calling DosSetNPHState. A call to DosOpen fails if all instances of the named pipe are already open.

The open also fails if the pipe has been closed by a client, but the server has not called DosDisConnectNPipe (to acknowledge the client's close), followed by DosConnectNPipe (to prepare the pipe for the next client). In both of these situations, ERROR_PIPE_BUSY is returned.

If all instances of a named pipe are busy, a client process can call DosWaitNPipe to wait for an instance to become available before it calls DosOpen again.

After a pipe instance has been opened by a client, that same instance cannot be opened by another client at the same time. However, the opening process can duplicate the handle as many times as desired by calling DosDupHandle. This enables child processes to share access to a pipe instance with a parent process.

The access-mode and sharing-mode fields that are specified for DosOpen must be the same as those that were specified by the server when it created the pipe with DosCreateNPipe.

Reading from Named Pipes

Both the server and the client processes read data from a pipe by calling DosRead. The server reads from the handle that was returned when it created the pipe with DosCreateNPipe, and the client reads from the handle that was returned to it by DosOpen.

When a pipe is created, the *PipeMode* parameter is used to specify both the *pipe type* and the *read mode* for the server end of the pipe:

- A byte pipe can be read-only in byte-read mode. (DosCreateNPipe and DosSetNPHState return ERROR_INVALID_PARAMETER if message-read mode is specified for a byte pipe.) In byte-read mode, all currently available data is returned, up to the buffer size specified by DosRead.
- A message pipe can be read in either byte-read mode or message-read mode, as follows:
 - When a message pipe is read in byte-read mode, the message headers are skipped, and the pipe is read as if it were a byte pipe.
 - When a message pipe is read in message-read mode, each message is read either in its entirety, or not at all, depending on the size of the message and the buffer length:
 -- If the buffer length that was specified for DosRead is larger than the next available message, then only that message is read, and the Bytes-Read parameter indicates the size of the message.
 -- If the buffer length for DosRead is smaller than the next available message, DosRead returns the number of bytes requested and ERROR_MORE_DATA. Subsequent calls to DosRead are blocked until the rest of the message can be transferred. (DosPeekNPipe can be used to find out how many bytes are left in the message.)

The *PipeMode* parameter of DosCreateNPipe also specifies the *blocking mode* for the server end of the pipe:

- If nonblocking mode was specified, DosRead returns immediately with 0 in the Bytes-Read parameter if no data is available.
- If blocking mode was specified, DosRead blocks until data is available; the only time it will return with 0 in the Bytes-Read parameter is if it reaches end-of-file.

DosRead works the same for both ends of the pipe. However, the read mode and blocking mode are not necessarily the same for the client end of the pipe as they are for the server end, because DosOpen always opens the CLIENT end in byte-read mode and blocking mode.

The read mode and blocking mode for either end of the pipe can be changed by calling DosSetNPHState. The pipe type, however, is always the same for both the server and client ends of the pipe, and it cannot be changed.

Writing to Named Pipes

Both the server and the client processes write to a pipe by calling DosWrite. The server writes to the handle that was returned to it by DosCreateNPipe, and the client writes to the handle that was returned to it by DosOpen.

Either bytes or messages can be written, depending on whether the pipe was created as a byte pipe or as a message pipe.

Named pipes created with the NP_ACCESS_OUTBOUND access mode cannot use the DosWrite function. If the named pipe's client uses the DosWrite function, the function returns error code 5 (ERROR_ACCESS_DENIED).

An attempt to write to a pipe whose other end has been closed returns ERROR_BROKEN_PIPE or, if the other end was closed without without reading all pending data, ERROR_DISCARDED.

When a process writes to a message pipe, the buffer-length parameter for DosWrite holds the size of the message that the process is writing. Because DosWrite automatically encodes message lengths in the pipe, applications do not have to encode this information in the data buffers.

The action taken by DosWrite depends on the blocking mode of the pipe, which is not necessarily the same for the server and client ends of the pipe. For the server process, the blocking mode of the pipe is specified when the pipe is created. For a client process, the blocking mode is automatically set to blocking when the pipe is opened. The blocking mode can also be reset by calling DosSetNPHState.

If the end of the message pipe that is being written to is in blocking mode, DosWrite does not return until all of the requested bytes have been written. (It might have to wait for the first part of the message to be read before it can write the rest of the message.)

If the message pipe is in nonblocking mode, DosWrite takes the following action:

- If the message is larger than the pipe buffer, DosWrite blocks until the entire message has been written. (Again, it might have to wait for the first part of the message to be read before it can write the rest of the message.)
- If the message is smaller than the pipe buffer, but there is currently not enough room in the buffer, DosWrite returns with a value of 0 in the Bytes-Written parameter.

If a byte pipe is in nonblocking mode, and if there is more data to be written than will fit in the pipe buffer, then DosWrite writes as many bytes as will fit in the buffer and returns the number of bytes that were actually written.

If a process tries to write to a pipe whose other end is closed, ERROR_BROKEN_PIPE is returned.

Synchronizing Named Pipe Dialogs

Communicating processes can synchronize their named pipe dialogs by calling DosResetBuffer after each call to DosWrite.

When used with external files, DosResetBuffer flushes the buffer cache of the requesting process to disk. When used with named pipes, this function blocks the calling process at one end of the pipe until the data the calling process has written to the pipe has been successfully read at the other end of the pipe.

Determining Pipe Status

DosQueryNPHState and DosQueryNPipeInfo can be used to obtain information about named pipes.

DosQueryNPHState
A client process can read data from the pipe, write data to the pipe, or both, depending on the access mode specified when the pipe was created. To check the current access mode, the client process can call DosQueryNPHState.

The following code fragment shows how to use DosQueryNPHState to obtain information about a named pipe:

```
#define INCL_DOSNMPIPES   /* Named-pipe values */
#include <os2.h>
#include <stdio.h>

HPIPE    hpHandle;            /* Pipe handle       */
ULONG    ulPipeHandleState;   /* Pipe-handle state */
APIRET   ulrc;                /* Return code       */

ulrc = DosQueryNPHState(hpHandle, &ulPipeHandleState);

if (ulrc != 0) {
    printf("DosQueryNPHState error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, *PipeHandleState* will contain information that describes the nature of the named pipe.

DosQueryNPHState returns the following information about the pipe handle and the attributes of the pipe:

- The end of the pipe that the handle is for (server or client end)
- The pipe type (byte pipe or message pipe)
- The instance count
- The blocking mode (blocking or nonblocking)
- The read mode (byte-read mode or message-read mode)

The values for pipe type and instance count cannot be changed, so they are always the same as those that were specified when the pipe was created with DosCreateNPipe. The information returned for blocking mode and read mode, however, can come from different sources:

- If the handle is for the server end of the pipe, then the blocking mode and the read mode were set with DosCreateNPipe, but could have been reset with DosSetNPHState.
- If the handle is for the client end of the pipe, then the blocking mode and the read mode were set to "blocking" and "byte-read" by the system when the client called DosOpen. However, again, they could have been reset with DosSetNPHState.

The pipe attributes are described in more detail in Creating Named Pipes.

An application can use DosSetNPHState to change the wait mode and the read mode. The pipe cannot be changed to no-wait mode when another thread is blocked on a read or write operation to the same end of the pipe.

DosQueryNPipeInfo
More detailed information about a named pipe can be obtained by using DosQueryNPipeInfo. This function returns information in a PIPEINFO data structure that includes the name of the pipe, the current and maximum instance counts (the current number of pipes and the maximum number of times the pipe can be created), the size of the input and output buffers for the pipe, and the pipe identifier of the client process.

The following code fragment shows how to use DosQueryNPipeInfo:

```
#define INCL_DOSNMPIPES   /* Named-pipe values */
#include <os2.h>
#include <stdio.h>

HPIPE    hpHandle;          /* Pipe handle                     */
ULONG    ulInfoLevel;       /* Pipe data required              */
PIPEINFO pipInfoBuf;        /* Pipe information data structure */
ULONG    ulInfoBufSize;     /* Pipe data-buffer size           */
APIRET   ulrc;              /* Return code                     */

ulInfoLevel = 1;                    /* Ask for standard level of pipe info */

ulInfoBufSize = sizeof(PIPEINFO);   /* Length of pipe info data structure  */

ulrc = DosQueryNPipeInfo(hpHandle,
                         ulInfoLevel,
                         &pipInfoBuf,
                         ulInfoBufSize);

if (ulrc != 0) {
    printf("DosQueryNPipeInfo error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the pipe information data structure contains a set of information describing the nature and the current state of the named pipe.

DosQueryNPipeInfo returns level 1 or level 2 file information for the pipe. Level 1 information includes the following:

- The actual sizes of the in-buffer and out-buffer
- The maximum number of pipe instances permitted
- The current number of pipe instances
- The length of the pipe name

- The ASCIIZ name of the pipe, including *ComputerName* if the pipe is in a remote computer system.

 Level 2 information consists of a unique 2-byte identifier for each of the pipe's client processes.

Examining the Contents of Named Pipes

DosPeekNPipe examines the current contents of a named pipe.

Named pipes created with the NP_ACCESS_INBOUND access mode cannot use the DosPeekNPipe function. If the named pipe's client uses the DosPeekNPipe function, the function returns error code 5 (ERROR_ACCESS_DENIED).

It is similar to DosRead, except that DosPeekNPipe does not remove data from the pipe. In addition, DosPeekNPipe never blocks, even if the pipe is in blocking mode; if the pipe cannot be accessed immediately, ERROR_PIPE_BUSY is returned.

Because DosPeekNPipe does not block, it returns only what is currently in the pipe. Thus, if a message pipe is being examined, only a portion of a message might be returned, even though the specified buffer length could accommodate the entire message.

DosPeekNPipe also returns the state of the pipe. A named pipe can be in any of the following states: Connected, Disconnected, Listening, Closing.

The following code fragment shows how to use DosPeekNPipe:

```
#define INCL_DOSNMPIPES   /* Named-pipe values */
#include <os2.h>
#include <stdio.h>

HPIPE              hpHandle;      /* Pipe handle                */
UCHAR              ucBuffer[200]; /* Address of user buffer     */
ULONG              ulBufferLen;   /* Buffer length              */
ULONG              ulBytesRead;   /* Bytes read (returned)      */
struct _AVAILDATA  BytesAvail;    /* Bytes available (returned) */
ULONG              ulPipeState;   /* Pipe state (returned)      */
APIRET             ulrc;          /* Return code                */

ulBufferLen = 200;  /* Length of the read buffer */

ulrc = DosPeekNPipe(hpHandle,
                    ucBuffer,
                    ulBufferLen,
                    &ulBytesRead,
                    &BytesAvail,
                    &ulPipeState);

if (ulrc != 0) {
    printf("DosPeekNPipe error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the input buffer *Buffer* will contain up to the first 200 bytes from the named pipe, *BytesRead* will contain the number of bytes read into *Buffer* , *BytesAvail* will contain the total number of bytes that were available in the pipe, and *PipeState* will contain a value indicating the state of the named pipe.

Closing Named Pipes

DosClose closes the specified pipe handle. When all of the handles that access one end of a pipe have been closed, the pipe is referred to as a *broken pipe* .

If the client end of the pipe closes, no other process can reopen the pipe until the server calls DosDisConnectNPipe (to acknowledge the client's close) followed by DosConnectNPipe (to prepare the pipe for a new client). Until it calls DosDisConnectNPipe, the server will receive ERROR_EOF if it tries to read from the pipe, and ERROR_BROKEN_PIPE if it tries to write to it. Clients that attempt to open the pipe receive ERROR_PIPE_BUSY.

If the server end closes when the client end is already closed, the pipe is deallocated immediately; otherwise, the pipe is not deallocated until

the last client handle is closed.

The following code fragment shows how to close a named pipe. Assume that a previous call to DosOpen provided the named pipe handle that is contained in *Handle* .

```
#define INCL_DOSNMPIPES   /* Named-pipe values */
#include <os2.h>
#include <stdio.h>

HPIPE   hpHandle;      /* Pipe handle */
APIRET  ulrc;          /* Return code */

ulrc = DosDisConnectNPipe(hpHandle);

if (ulrc != 0) {
    printf("DosDisConnectNPipe error: return code = %ld",
           ulrc);
    return;
}
```

Program Execution Control

Multitasking is the ability of OS/2 to manage the execution of more than one application at a time. A multitasking operating system, such as OS/2 enables users to run many applications simultaneously.

For the programmer, OS/2 supports two types of multitasking. An application can start other programs, in separate processes, that will execute concurrently with the application. These programs can be a new copy of the application, a related program that is designed to work with the application, or an unrelated program. Running multiple processes is the first type of multitasking provided for programmers.

Running multiple threads is the second type of multitasking supported by OS/2. OS/2 enables applications to run multiple threads of execution within the same process; separate activities can be multitasked within the application. An example of multiple-thread multitasking would be for the application to dispatch a separate subroutine to load a file from the disk, and have the subroutine execute at the same time the main program continues to monitor and respond to user input.

This chapter describes processes, threads, and sessions, and the OS/2 functions used to create and manage them. Additionally, there is a section describing CPU scheduling.

The following topics are related to the information in this chapter:

- Memory
- Semaphores
- Queues
- Pipes
- Exception handling
- Debugging

About Program Execution Control-Thread, Processes, and Sessions

To successfully use multitasking-multiple processes and multiple threads-in your programs, you need to understand the difference between a thread, a process, and a session.

A *thread* is a dispatchable unit of execution that consists of a set of instructions, related CPU register values, and a stack. Each process has at least one thread, called the *main thread* or *thread 1* , and can have many threads running simultaneously. The application runs when OS/2 gives control to a thread in the process. The thread is the basic unit of execution scheduling.

A *process* is the code, data, and other resources-such as file handles, semaphores, pipes, queues, and so on-of an application in memory. OS/2 considers every application it loads to be a process. System resources are allocated on a per-process basis.

A *session* is one (or more) processes with their own virtual console. (A virtual console is a virtual screen-either a character-based, full screen or a Presentation Manager window-and buffers for keyboard and mouse input.)

OS/2 supports up to 255 concurrent sessions and up to 4095 processes. OS/2 supports a system-wide maximum of 4095 threads, but the

number of threads available in a single process will be lower, and will vary, because of resource usage within the process.

Threads

Applications always have at least one thread of execution-thread 1. Using multiple threads of execution, an application can do several things at the same time.

For example, a simple Presentation Manager application consists of a single process with two threads:

- A user interface thread that listens and responds to user requests, and that queues work for the second thread
- A processing thread that handles lengthy processing

OS/2 creates the first thread of execution for a process when it starts the executable file. To create another thread of execution, a thread calls DosCreateThread, specifying the address within the program module where the thread begins asynchronous execution. Although a thread can execute any part of the application, including a part being executed by another thread, threads typically are used to execute separate sections of the application. By using several threads, the system can distribute the available CPU time and enable an application to carry out several tasks simultaneously. For example, an application can load a file and prompt the user for input at the same time.

Each thread in a process has a unique stack and register context. Threads shares the resources of the process with the other threads in the process. For example, threads in the same process have access to the memory spaces of other threads within the process. However, threads of one process do not have access to the data spaces of other processes.

Each thread has a priority, that determines the amount of CPU time the thread is allocated. Threads inherit the priority of the thread that creates them. The priority of a thread can be changed by the application; see Changing the Priority of a Thread for details.

An application can use DosSuspendThread and DosResumeThread to suspend and resume the execution of a given thread. When an application suspends a thread, the thread remains suspended until the application calls DosResumeThread.

When an application has more than one thread, it might be necessary to ensure that one thread is finished executing before another thread uses a shared resource, such as a disk file. DosWaitThread causes the application to wait until a specific thread has finished. DosWaitThread can also be used to determine the state of a thread; the function can return immediately with an error value if the identified thread is still running.

A thread ends when it calls DosExit.

Processes

An OS/2 application that has been loaded into memory and prepared for execution is called a process. As mentioned earlier, a process consists of the code, data, and other resources (for example, open file handles) that belong to the application. Each process has at least one thread, called the *main thread* or *thread 1* .

When OS/2 runs an application, it confirms that the process code and data are in memory and that the main thread's registers and stack are set before starting the application. Each application has access to all resources of the computer, such as memory, disk drives, screen, keyboard, and the CPU itself. The system carefully manages these resources so that applications can access them without conflict.

A process can have more than one thread. OS/2 creates the first thread of execution for a process when it starts the executable file. More threads can be created with DosCreateThread. Each thread runs independently, with its own stack and register values. Unless the application changes a thread's priority, each thread gets a slice of the CPU in a round-robin strategy. All the threads in a process share the application's globally defined variables and other resources (open file handles, and so on).

A process or thread ends when it calls DosExit. OS/2 automatically closes any files or other resources left open by the process when the process ends. When a thread ends, however, any open resources remain open until another thread closes them or the process ends. A process can direct OS/2 to carry out other actions when the process ends, by using DosExitList to create a list of termination functions. OS/2 calls the termination functions, in the order given, when the process is about to end. If the thread has registered any exception handlers, the exception handlers will also be called before the thread ends.

Creating Processes

An application can load and execute other applications by using DosExecPgm. The new application, once loaded, is called a *child process* . The process that starts the new application is called the *parent process* .

A child process is like any other process. It has its own code, data, and threads. The child process inherits the resources-such as file handles, pipes, and queues-that belong to the parent process at the time the child process is created, although not necessarily with the same access rights. The parent process can place restrictions on the access of the child process to these resources:

- Files are inherited except for files that were opened with no inheritance indicated.
- Pipes are inherited.

Assuming that the parent process gives the child process appropriate access rights, the child process can use the inherited resources without preparing them. For example, if the parent process opens a file for reading and then starts a child process, the child process can read from the file immediately; it does not have to open the file. However, once the child process is created additional resources that the parent process creates are not available to the child process. Similarly, resources the child process creates are not available to the parent process.

The parent process also has control over the meanings of standard input, output, and error for the child process. For example, the parent can write a series of records to a file, open the file as standard input, open a listing file as standard output, and then execute a sort program that takes its input from standard input and writes to standard output.

Note that memory is not included in the list of things that a child process can inherit from its parent process. The child process is created with its own process address space that is separate and distinct from the memory of the parent process. A new linear address space is built for the new process. The only way for a parent process and a child process to access the same memory is to set up a shared memory area.

The executable file of the child process can be started either synchronously or asynchronously to the parent process. If the parent process starts the child process running *synchronously* , the parent process is suspended and waits until the child process ends before continuing. A child process running *asynchronously* executes independently of the parent process (that is, both run at the same time). The parent process specifies how the child process is to run by setting a parameter in the call to DosExecPgm.

The OS/2 command processor, CMD.EXE, starts most child processes synchronously. The parent process waits for each child process to end before it prompts the user for the next command. The command processor also enables the user to start asynchronous applications by using the DETACH command. When the user detaches an application, the command processor starts the application asynchronously, in the background, and continues to prompt for input.

Process Termination

A parent process can use DosWaitChild to determine the termination status of a child process that is running independently. The parent process can have one of its threads call DosWaitChild to wait for completion of the child process while other threads of the parent continue processing.

If the child has started another process, DosWaitChild waits for the completion of any grandchild processes before returning, but does not report their status. If the specified child process has multiple threads, DosWaitChild returns the result code of the last DosExit request.

If there are no child processes, either active or ended with a return code, DosWaitChild returns with an error code. If no child processes have ended, DosWaitChild can optionally wait until one ends before returning to the parent.

Process Exit Lists

Because any process can end any other process for which it has a process identifier, applications might lose information if a process ends the application before it can save its work. To prevent this loss of data, you can create a list of functions to clean up data and files before OS/2 ends the process. This list is called an *exit list* . OS/2 maintains an exit list for each process and calls these functions whenever the application is ended, whether by another process or by itself.

You call DosExitList to add to the exit list a routine that is to be given control when a process is ended (or finishes its execution). Multiple routines can be added to the list. When the process is ending, OS/2 transfers control to each address on the list.

Exit-list functions perform clean-up operations on resources. For example, an exit-list function can be used in a dynamic link library module to free resources or clear flags and semaphores when a client program has ended.

Multitasking with Threads and Multitasking with Processes

The creation and termination of a process is relatively slow compared to the creation and termination of a thread, and is more costly in terms of system resources.

For example, sharing data and resources between processes requires shared memory and the mechanisms of interprocess communication; threads, on the other hand, have full access to the memory and other resources that belong to the process the threads are part of and can be coordinated using semaphores. For these reasons, thread-to-thread task context switches are faster than process-to-process context switches.

Because OS/2 can create and execute threads more quickly than processes, the preferred multitasking method for applications is to distribute tasks among threads in the same process instead of among processes.

Sessions

OS/2 uses sessions to help the user move from one application to the next without disrupting the screen display of an application.

A *session* consists of at least one process and a virtual console-buffers for keyboard and mouse input and either a character-based, full screen or a Presentation Manager window. When the system creates a session, the process in the session displays output in the screen or window. The user can view the output and supply input by moving to the session. The user moves to a session by pressing the Alt+Esc key combination, by selecting the title of the session from the Window List, or, for windowed sessions, by clicking the mouse in the session window.

A *child session* is under the control of the session that creates it. The session that starts the child session is called the *parent session* . Any process in the parent session can exercise control over a child session.

An *unrelated session* is not under the control of the session that started it. The process that creates the unrelated session cannot select it, make it nonselectable, bind it, or end it, nor can any other session. DosStartSession does not even return a session identifier when an unrelated session is started. Unrelated sessions are controlled entirely by the user. When OS/2 starts new sessions, it starts them as unrelated sessions.

Creating Sessions

A process creates a new session by using DosStartSession. DosStartSession enables an application to start another session and to specify the name of the application to be started in that session.

DosStartSession also specifies which of the five session types is to be started:

- Full screen, protect mode
- Text windowed, protect mode
- Presentation Manager (PM)
- Full screen DOS Session
- Windowed DOS Session

Protect mode applications run in full screen and text windowed sessions, PM and AVIO applications run in PM windows, and DOS applications run in full screen DOS Sessions and windowed DOS Sessions.

OS/2 applications running in any of the OS/2 session types-full screen, text windowed, and PM-can start sessions of any other type, including DOS Sessions. DOS Session applications cannot start other sessions.

An application can start another process in a separate session when the application will not manage any I/O done by the process. For example, an application that starts an unrelated application could start it in a separate session.

A session can be started as a related or an unrelated session. A related session is called a *child session* , and the session starting the child session is called the *parent session* . An application can control its child sessions by using the session identifier returned by DosStartSession with the DosSetSession, DosSelectSession, and DosStopSession. If an application starts an *unrelated session* , the new session cannot be controlled by the application. The Related field in the STARTDATA structure specifies whether the new session is related to the session calling DosStartSession.

After a process has started a child session, no other process in its session can start a child session until all dependent sessions started by this process have ended.

When a session is created, the title specified in the function call (or the application name if no title is specified) is added to the Window List.

DosStartSession can be used to start either a foreground or a background session, but a new session can be started in the foreground only when the caller's session, or one of the caller's descendant sessions, is currently executing in the foreground. The foreground session for windowed applications is the session of the application that owns the window focus.

Termination Queues
The parent session must create a *termination queue* prior to specifying the queue name in a call to DosStartSession. OS/2 will continue to notify the parent session through the specified queue as long as the session calling DosStartSession remains a parent session. In other

words, when all the child sessions for a particular parent session end, the termination queue is closed by OS/2. An existing queue name must be specified on the next DosStartSession call if the caller wants to continue receiving termination notification messages.

OS/2 writes a data element to the specified queue when any child session ends. The queue is posted regardless or who ends the child session (the child, the parent, or the user) and whether the termination is normal or abnormal.

A parent session calls DosReadQueue to receive notification when a child session has ended. The word that contains the request parameter, returned by DosReadQueue, will be 0. The data element has the following format:

Termination Queue Element Format

```
Size       Description

WORD       Session ID of the child session that ended

WORD       Result code
```

The process that originally called the DosStartSession request should call DosReadQueue, with the NOWAIT parameter set to 0. This is the only process that has addressability to the notification data element. After reading and processing the data element, the caller must free the segment containing the data element by calling DosFreeMem.

When a child session ends, the result code returned in the termination queue's data element is the result code of the program specified in the DosStartSession call, providing either one of the following is true:

- The program was run directly, with no intermediate secondary command processor
- The program is run indirectly through a secondary command processor, and the /C parameter is specified

When a child session is running in the foreground at the time it ends, the parent session becomes the new foreground session. When a parent session ends, any child sessions are ended. When an unrelated session ends in the foreground, OS/2 selects the next foreground session.

Child Session Control

A session can be either a child session or an unrelated session. A child session is under the control of the processes in the session that creates it (the parent session). A process can select, set, or stop a child session by using DosSelectSession, DosSetSession, or DosStopSession, respectively. DosStartSession returns a unique session identifier for the child session for use in these functions.

A session can run in either the foreground or background. A process can create a foreground session only if the creating process or one of its descendant sessions is executing in the current foreground session. A process can move a child session to the foreground by selecting the child session using the session identifier and calling DosSelectSession. A process can make a child session nonselectable by using DosSetSession to change the SelectInd field in the STATUSDATA structure. This prevents the user from selecting the session from the Window List but does not prevent a process from selecting the child session by using DosSelectSession.

A process can bind a child session to its own session by using DosSetSession. Binding a session causes that session to move to the foreground whenever the user selects the parent session from the Window List.

A parent session can use a session identifier with the DosSetSession function only if the parent session created the child session associated with that identifier. It cannot use identifiers for child sessions created by other parent processes. This is true for all session management functions.

Although a child session is related to the session that started it, the processes in the child and original sessions are not related. This means that even though DosStartSession supplies the process identifier of the process in the child session, the process identifier cannot be used with OS/2 functions such as DosSetPriority.

Child Session Termination

A parent session can stop a child session by using DosStopSession. Stopping the child session ends the processes in that session. It also stops any sessions related to the child session. If a child session is in the foreground when it is stopped, the parent session becomes the foreground session. DosStopSession breaks any bond that exists between the parent session and the specified child session.

A process running in the session specified in the call to DosStopSession can ignore the request to end. If this happens, DosStopSession still returns 0 (indicating success). The only way to be certain that the child session has ended is to wait for notification through the termination queue specified in the call to DosStartSession. OS/2 writes a data element into the specified queue when the child session ends. The process

in the parent session must call DosReadQueue to retrieve this data element, which contains the session identifier for the child session and the return value for the process in the child session. Only the process that created the child session can read the data element.

About CPU Scheduling

OS/2 performs prioritized, preemptive, multitasking. *Prioritized* means that OS/2 does not divide CPU time equally among all threads. All programs do not get equal access to the CPU. A prioritizing, time-slicing strategy is used to allocate access to the CPU among competing threads. Each thread has a priority and OS/2 runs the highest priority thread that is ready to run. Programs with higher priorities (a real-time robotic application, for example), are given access to the CPU before programs with lower priorities. If a thread with a higher priority than the currently running thread becomes ready to run, the current thread is stopped immediately, or *preempted*, and the higher priority thread is given the CPU. The lower priority thread does not get to complete its time slice. Threads of equal priority are given CPU time in a round-robin manner.

Preemptive means that the multitasking activity needs no cooperation from the executing programs. OS/2 maintains control over executing programs, and stops, or preempts, them when their time slice with the CPU is over or when a higher priority program is ready to run.

CPU scheduling is based on four priority classes-Time Critical, Fixed-High, Regular, and Idle-Time. Each class has 32 levels of execution ordering. Scheduling parameters are user-selectable at the time the system is started or can be varied dynamically based on system load.

Depending on a thread's priority class and level, OS/2 periodically gives each thread in each process a small slice of CPU time. Threads with higher priorities always run before threads having lower priorities. A thread runs until its time is up or until a thread with a higher priority is ready to run. At that time, OS/2 preempts the thread and starts another thread. Threads can also voluntarily relinquish the CPU (for example, by calling DosSleep).

The amount of time in each time slice is defined by the TIMESLICE command in the CONFIG.SYS file. The TIMESLICE command can be used by the user to customize the size of the time slices that a thread gets. The default is for OS/2 to dynamically vary the size of the time slice based on the activity of the thread and the overall system load.

When a thread is created (using DosCreateThread), it inherits the priority of the thread that started it. DosSetPriority enables threads to change their priority classes and levels in response to changes in their execution environments. DosSetPriority enables a thread to change its own priority, or the priority of any thread within its process. DosSetPriority also enables changing priorities for the entire process and for descendant processes. Within each class, the priority level of a thread can vary because of a DosSetPriorty request or, if dynamic priority variation is being used, because of action taken by OS/2.

Priority Classes

OS/2 uses four priority classes to determine when a thread receives a time slice:

Priority Classes

Priority	Description
Time-critical	Highest priority. For use when response time is critical.
Fixed-high	Used by threads that provide service to other threads. This priority class is to be used when it is desirable that the thread not be too sensitive to the foreground/background boost provided by dynamic priority variation. It is meant for programs that need to execute before regular programs, but without the immediate response time requirement called for by time-critical threads.
Regular	Default priority. Most threads belong in this class.
Idle-time	Lowest priority. This priority only gets CPU time when there is no other work to

do.

A time-critical thread always receives a time slice before a fixed-high thread, a fixed-high thread always receives a time slice before a regular thread, and a regular thread always receives a time slice before an idle-time thread.

Time-Critical Threads
Time-critical threads have the highest priority class and execute before any fixed-high, regular, or idle-time threads.

The time-critical class is for threads that must react to events outside the system. For example, in a communications application, a thread responsible for reading data from the communications device needs enough time to read all incoming data. Because more than a regular time slice might be needed, a time-critical classification ensures that the thread gets all the time required.

Time-critical threads have a static priority that is not varied by OS/2. They are scheduled among themselves in priority level order, with round-robin scheduling of threads of equal priority.

Time-critical threads must be executed quickly, then free the CPU for other work until another time-critical event occurs. This is important to maintain good interactive responsiveness to the user and enable communications and other time critical applications to run concurrently. The time-critical activity should, when possible, be in a thread separate from the rest of the application, to isolate and minimize the time spent processing at the time-critical level. A good rule of thumb is that a time-critical thread should consist of no more than about 20,000 assembly language instructions.

Fixed-High Threads
A *fixed-high thread* has a priority class that is lower than time-critical but executes before any regular or idle-time threads. This class of threads should be used to provide service for other threads where it is desirable that the thread not be too sensitive to the foreground/background boost provided by dynamic priority variation. A messaging thread, would be a good example of this type of thread.

OS/2 varies the priority of a fixed-high thread around a base value according to the activity of the thread and the system at any point in time. The base value can be set by the thread itself.

Regular Threads
A *regular thread* is the class that the majority of threads fall into. No explicit action is necessary by the application to run at this priority, it is the default.

OS/2 varies the priority level of a regular thread around a base value according to the activity of the thread and the system at any point in time. The base value can be set by the thread itself.

Idle-Time Threads
An *idle-time* thread is one with very low priority that executes only when there are no regular, fixed-high, or time-critical threads to execute. Idle-time threads get CPU time only when there is no other work to do. The *idle-time* class is for threads that need very little CPU time.

Idle-time threads have a static priority that is not varied by OS/2.

Priority Levels

Within each class, OS/2 maintains a priority level for a thread. For each of the four priority classes, there are 32 priority levels-0 to 31. A thread with priority level 31 always receives a time slice before a thread with priority level 30, and so on.

If two or more threads have the same priority level, OS/2 distributes the CPU time equally by using a *round-robin scheme* ; that is, OS/2 gives a time slice to first one, then another, and so on, and then goes back to the first. A thread can use DosSetPriority to change its own priority or the priority of any other thread within its process.

Although an application can set the priority level of a thread at any time, only applications that use more than one thread or process should do so. The best use of priority is to speed up threads or processes on which other threads or processes depend. For example, an application might temporarily raise the priority of a thread loading a file if another thread is waiting for that file to be loaded. Because the priority of a thread is relative to all other threads in the system, raising the priority of the threads in an application merely to get the extra CPU time adversely affects the overall operation of the system.

There are other ways to affect the amount of CPU time a thread is given. A thread can define a critical section of code by using DosEnterCritSec and DosExitCritSec. While inside the critical section of code, a thread cannot be preempted by any other thread within its process (threads in other processes are still given their time slices). Using critical sections enables threads to get more CPU time, while not unduly affecting the rest of the system.

The priority class and priority level are set using DosSetPriority. The priority class is changed by passing the new priority class to the function. The priority level, however, is changed by passing a value, called the *priority-delta* , that is added to the existing priority level to produce the new priority level; changes to the priority level are relative to the current priority level. Specifying a positive priority-delta increases the priority level, enabling the thread to obtain more CPU time than it normally would. A negative priority-delta decreases the priority level, giving the thread less CPU time than it would normally receive. The value is restricted to the valid range, based upon the current priority class of the process.

If you change the priority level without changing the priority class, the priority-delta is relative to the current priority level. However, if you change the priority class at the same time that you change the priority level, the priority-delta value is relative to 0. Whenever DosSetPriority is called with a class specification, but no value is specified for priority-delta, the base priority level defaults to 0.

The process identifier parameter of DosSetPriority specifies which process is affected by the call. A process can change the priority of itself, of any process that is a descendant of itself, or of one of its threads.

A thread can change the priority of any thread within its current process. When a thread changes the priority of threads in a descendant process, however, only those threads running at the default priority will be changed. You cannot change the priority of a thread in a child process that has already changed its priority from the default.

The initial thread of execution for an application is given a regular class priority that varies by the system. When a thread is created, it is initially dispatched in the same class and priority as the thread that started it. A child process inherits the priority of the thread in the parent process that creates it.

Priority Guidelines

Within the two most common priority classes-time-critical and regular-there are certain broad guidelines recommended for choosing the priority level for a program.

- TIME-CRITICAL CLASS

 The guidelines for level within the time critical class are set to maximize the number of different applications that can successfully multitask in an OS/2 system. The guidelines are described in the following table.

 Recommended Priority Levels for Time-Critical Threads

Activity	Range of Recommended Priority Levels	Justification/Comments
Robotics/Real time process control	20-31	OS/2 systems can be used on manufacturing lines to control equipment or in other real time process control applications. In this case a slow response from the PC could cause expensive damage to equipment or even human injury. Therefore the highest priority levels should be reserved for these applications.
Communications	10-19	In communications, the inability to get the processor could cause a loss of data or communications sessions. Therefore this class of applications is next highest.
Other	0-09	Other threads might need to preempt the foreground in special cases (for example, Control-Break). These should be set below the other 2 classes.

 In general, application performance is not a good reason to make a thread time critical.

- REGULAR CLASS

 In cases where explicit priority levels are set, they should follow the guidelines listed below.

 Recommended Priority Levels for Regular Priority Threads

Activity	Range of Recommended Priority Level	Justification
Communications	26-31	Communications should take priority over other background processing to increase overlap with transmission and processing on the partner PC or host. This gives the best system performance.
Other	0-25.	If an application has multiple threads it might be necessary to set them to several different priorities to optimize the order in which they run. A range of priority levels is provided to facilitate this. (The default priority is regular class, level = 0.)

Dynamic Priority Alteration

OS/2 can be configured to dynamically alter the priority of a process. The PRIORITY statement in CONFIG.SYS can be set to either ABSOLUTE or DYNAMIC. If PRIORITY specifies the ABSOLUTE option, all processes receive CPU time based on the priority established by calls to DosSetPriority. If the PRIORITY command in the CONFIG.SYS file specifies the DYNAMIC option, OS/2 adjusts process priorities based on system load and process activity, and on whether the process is in the foreground. DYNAMIC is the default setting; if the PRIORITY command is not specified, the system uses DYNAMIC priority. DYNAMIC is designed to gives the best overall system performance under most conditions.

When dynamic priority variation is enabled, OS/2 grants higher priority to the foreground process than to background processes. System load and process activity are also taken into consideration. The priority of the process consists of a computed priority value that is based upon the display status of the process (foreground or background), and its recent I/O and CPU time usage history. When dynamic priority variation is enabled, I/O priority boosts are generated for keyboard input, window, foreground, processor starvation, device I/O, and DOS Session interrupts. This ensures that the foreground process-the process most likely to be in use-receives enough CPU time to provide quick response to user input.

There are times when dynamic priority variation can interfere with a multi-threaded application's execution. For example, if you are doing a lot of keyboard input on a thread, its priority will get boosted and other threads may not get enough CPU time. A communication thread is an example of a time sensitive background thread which would be one case. The solution is to either set PRIORITY = ABSOLUTE or to call DosSetPriority on a regular basis to keep the threads priority at the desired level.

Altering the Size of the Time Slice

The TIMESLICE command in CONFIG.SYS sets the minimum and maximum amount of processor time allocated to processes and programs for both OS/2 and DOS sessions.

The first parameter selects the minimum TIMESLICE value in milliseconds. This value is the minimum amount of time a thread is to be processed before yielding the processor to a thread of the same priority level. This value must be an integer greater than or equal to 32.

The second parameter selects the maximum TIMESLICE value in milliseconds. The value of this parameter is the maximum amount of time a thread can be processed before yielding processor time. This value must be an integer greater than or equal to the minimum value, and less than 65536.

The default is *dynamic time slicing* , which varies the size of the time slice depending on system load and paging activity. Dynamic time slicing is designed to give the best performance in most situations.

Using Processes

An OS/2 application that has been loaded into memory and prepared for execution is called a process. A *process* is the code, data, and other resources of the application, such as the open file handles, semaphores, pipes, queues and so on. OS/2 considers every application it loads to be a process.

Each process has at least one thread, called the *main thread* or *thread 1* . The application runs when the system scheduler gives control to a thread in the process. For more on thread management, see Using Threads.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Starting a Child Process

You start a process by calling DosExecPgm. The process you start is a child of the calling, or parent, process and inherits many of the resources owned by the parent process, such as file handles.

DosExecPgm creates a process environment from an executable file. The target program is loaded into storage, and it begins execution.

The following code fragment starts an application named ABC:

```
    #define INCL_DOSPROCESS       /* Process and thread values */
    #include <os2.h>

    CHAR szFailName[CCHMAXPATH];
    RESULTCODES resc;

    DosExecPgm(szFailName,           /* Object-name buffer  */
               sizeof(szFailName),   /* Length of buffer    */
               EXEC_SYNC,            /* Sync flag           */
               (PSZ) NULL,           /* Argument string     */
               (PSZ) NULL,           /* Environment string  */
               &resc,                /* Address for result  */
               "ABC.EXE");           /* Name of application */
```

In this example, ABC runs synchronously (as specified by EXEC_SYNC). This means the parent process temporarily stops while the child process runs. The parent process does not continue until the child process ends.

Starting an Asynchronous Child Process

To start a child process and enable it to run asynchronously (that is, without suspending the parent process until the child process ends), you use the EXEC_ASYNC constant in a call to DosExecPgm. If you start a process in this way, the function copies the process identifier of the child process to the *codeTerminate* field of the RESULTCODES structure that is returned by DosExecPgm. You can use this process identifier to check the progress of the child process or to end the process.

You can also run a child process asynchronously by using DosExecPgm with the EXEC_ASYNCRESULT constant. In addition to causing DosExecPgm to return to the parent process immediately, this constant also directs OS/2 to save a copy of the termination status when the child process ends. This status specifies the reason the child process stopped. The parent process can retrieve the termination status by using DosWaitChild.

The following code fragment starts the program SIMPLE.EXE, and then waits for it to finish. It then prints the termination code and the return code.

```
    #define INCL_DOSPROCESS       /* Process and thread values */
    #include <os2.h>
    #include <stdio.h>

    #define START_PROGRAM "SIMPLE.EXE"

    CHAR         szLoadError[100];
    PSZ          pszArgs;
    PSZ          pszEnvs;
    RESULTCODES  rcReturnCodes;
    APIRET       ulrc;

    ulrc = DosExecPgm(szLoadError,         /* Object name buffer           */
                      sizeof(szLoadError), /* Length of object name buffer */
                      EXEC_SYNC,           /* Asynchronous/Trace flags     */
                      pszArgs,             /* Argument string              */
                      pszEnvs,             /* Environment string           */
                      &rcReturnCodes,      /* Termination codes            */
                      START_PROGRAM);      /* Program file name            */

    printf("Termination Code %d  Return Code %d \n",
           rcReturnCodes.codeTerminate,
           rcReturnCodes.codeResult);

/*----------------SIMPLE.EXE------------------*/

    #define INCL_DOSPROCESS       /* Process and thread values */
    #include <os2.h>
    #include <stdio.h>
```

```
#define RETURN_CODE 0

main()
{
    printf("Hello!\n");
    DosExit(EXIT_PROCESS,     /* End the thread or process */
            RETURN_CODE);     /* Result code               */
}
```

Starting a Background Process

You can start a child process in the background by specifying the EXEC_BACKGROUND constant in DosExecPgm. A background process runs independently of the parent process and is called a *detached process* . A detached process should not require any input from the keyboard or output to the video screen, but it can use interprocess communication, such as pipes, queues, and shared memory.

The following code fragment starts the program BATCH.EXE in the background.

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>
#include <stdio.h>

#define START_PROGRAM "BATCH.EXE"

CHAR         szLoadError[100];
PSZ          pszArgs;
PSZ          pszEnvs;
RESULTCODES  rcReturnCodes;
APIRET       ulrc;

ulrc = DosExecPgm(szLoadError,         /* Object name buffer           */
                  sizeof(szLoadError), /* Length of object name buffer */
                  EXEC_BACKGROUND,     /* Asynchronous/Trace flags     */
                  pszArgs,             /* Argument string              */
                  pszEnvs,             /* Environment string           */
                  &rcReturnCodes,      /* Termination codes            */
                  START_PROGRAM);      /* Program file name            */

if (ulrc != 0) {
    printf("DosExecPgm error: return code = %ld",
           ulrc);
    return;
}
```

Setting the Command Line and Environment for a Child Process

When you start a process, it inherits many of the resources of the parent. This includes file handles, such as the standard input and standard output files. A child process also inherits the resources of the screen group, such as the mouse and video modes, and the environment variables of the parent process.

The call to DosExecPgm determines the command line and environment that the child process receives. The fourth and fifth parameters of the function are pointers to the command line and the environment, respectively. If these pointers are NULL, the child process receives nothing for a command line and only an exact duplicate of the environment of the parent process. The parent process can modify this information by creating a string (ending with two NULL characters) and passing the address of the string to the function. The command line string must include the name of the application, followed by a NULL character, and the command line arguments, followed by two NULL characters. Any number of arguments can be passed to the child process, as long as the argument string ends with two NULL characters.

The following code fragment passes to the child process the string "test -option1 -option2" as its command line:

```
#define INCL_DOSPROCESS    /* Process and thread values */
```

```
#include <os2.h>

RESULTCODES resc;
CHAR szFailName[CCHMAXPATH];

CHAR szCommandLine[] = "test\0-option1 -option2\0";

DosExecPgm(szFailName,          /* Object-name buffer  */
           sizeof(szFailName),  /* Length of buffer    */
           EXEC_SYNC,           /* Sync flag           */
           szCommandLine,       /* Argument string     */
           (PSZ) NULL,          /* Environment string  */
           &resc,               /* Address of result   */
           "test.exe");         /* Name of application */
```

Changing the Priority of a Process

Changing the priority of a process is simply a matter of changing the priority of every thread executing in the process. For the details see the section on changing thread priorities, Changing the Priority of a Thread.

Obtaining Information about Child Processes

OS/2 creates and maintains a process information block for every process. An application can use DosGetInfoBlocks to access the process information block. This function returns a pointer to a *PIB* data structure, which contains the information from the process information block.

The following code fragment returns the address of the process information block of the current process. The calling thread can subsequently browse either the *PIB* block.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#include <os2.h>

PTIB     ptib;      /* Address of pointer to thread information block  */
PPIB     ppib;      /* Address of pointer to process information block */
APIRET   rc;        /* Return code                                     */

rc = DosGetInfoBlocks(&ptib,
                      &ppib);
```

DosGetInfoBlocks also returns the address of the thread information block of the current thread.

Waiting for a Child Process to End

You can synchronize the execution of a process with the execution of one of its child processes by calling DosWaitChild. DosWaitChild does not return until the specified child process ends. This can be useful, for example, if the parent process needs to ensure that the child process has completed its task before the parent process continues with its own task.

In the following code fragment, the parent process starts a child process and then waits for the child process to finish:

```
#define INCL_DOSPROCESS     /* Process and thread values */
#include <os2.h>

RESULTCODES resc;
PID pidEnded;
CHAR szFailName[CCHMAXPATH];

CHAR szCommandLine[] = "APP\0test\0";
```

```
DosExecPgm(szFailName,              /* Failed-name buffer     */
           sizeof(szFailName),      /* Length of buffer       */
           EXEC_ASYNC,              /* Sync flag              */
           szCommandLine,           /* Argument string        */
           (PSZ) NULL,              /* Environment string     */
           &resc,                   /* Address of result      */
           "APP.EXE");              /* Name of application    */

DosWaitChild(DCWA_PROCESS,          /* Only the process       */
             DCWW_WAIT,             /* Waits until it is done */
             &resc,                 /* Puts the result here   */
             &pidEnded,             /* PID of ended process   */
             resc.codeTerminate);   /* Child to wait for      */
```

You can cause a process to wait for all its child processes to end by using the DCWA_PROCESSTREE constant in the call to DosWaitChild.

Ending the Current Process

You end the current process by calling DosExit. When you exit, the system stops the process and frees any existing resources the process owns.

In the following code fragment, DosExit is used to end the process if the given file does not exist:

```
#define INCL_DOSPROCESS     /* Process and thread values */
#include <os2.h>

#define HF_STDERR 2         /* Standard error handle     */

HFILE   hf;
ULONG   ulAction, ulWritten;
APIRET  rc;

rc = DosOpen("SAMPLE.TXT",
             &hf,
             &ulAction,
             0,
             FILE_NORMAL,
             FILE_OPEN,
             OPEN_ACCESS_WRITEONLY |
             OPEN_SHARE_DENYWRITE,
             (PEAOP2) NULL);

if (rc) {
    DosWrite(HF_STDERR,
             "Cannot open file\r\n",
             18,
             &ulWritten);

    DosExit(EXIT_PROCESS,
            rc);
}
```

EXIT_PROCESS directs DosExit to end all the threads in a process including the calling thread, thus ending the process. DosExit includes an error value that is returned to the parent process through the RESULTCODES structure specified in the DosExecPgm call that started the process. If you started the application from the command line, the command processor, CMD.EXE, makes this value available through the ERRORLEVEL variable. If another process started the application, that process can call DosWaitChild to determine the error value.

If you want to exit only from a given thread, you can call DosExit with the EXIT_THREAD constant. This will end only the calling thread; other threads in the process are not affected. If the thread you end is the last thread in the process, the process also ends. If the thread consists of a function, the thread ends when the function returns.

Terminating a Process

A process can end the execution of a descendant process by calling DosKillProcess. This causes OS/2 to send a XCPT_SIGNAL_KILLPROC exception to the target process. The child processes of the target process can also be ended.

The following code fragment ends the specified process and all child processes belonging to that process:

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>

PID pidProcess;

DosKillProcess(DKP_PROCESSTREE,
               pidProcess);
```

In this example, the *pidProcess* parameter specifies which descendant process to end. The process identifier is returned by DosExecPgm in the *codeTerminate* field of the RESULTCODES structure when you start the child process.

The parameter DKP_PROCESSTREE in the example indicates that the specified process, *pidProcess* , and all of its descendant processes are to be ended.

If you specify DKP_PROCESS in a call to DosKillProcess, only the specified process is ended. Its child processes, if any, continue to run.

The process to be ended must either be the current process, or it must have been directly created by the current process with DosExecPgm for asynchronous execution. That is, a process can end itself and its descendants.

The process to be ended need not still be executing. If it has started its own child processes, but has stopped executing, its children can still be flagged for termination.

Obtaining the Termination Status of a Child Process

OS/2 saves the termination status for a process if the process was started by using the EXEC_ASYNCRESULT constant in the call to DosExecPgm.
You can retrieve the termination status of the most recently ended process by using the DCWW_NOWAIT constant with DosWaitChild and setting the child process identification parameter to 0. The DCWW_NOWAIT constant directs the function to return immediately, without waiting for a process to end. Instead, the function retrieves the termination status from the process that most recently ended. If you specify a child process identification with DCWW_NOWAIT, DosWaitChild returns ERROR_CHILD_NOT_COMPLETE if the child process has not ended. Once the specified process has ended, DosWaitChild returns its termination code.

The following code fragment starts a child session (the program SIMPLE.EXE), and then retrieves the termination status from the process that most recently ended.

```
#define INCL_DOSPROCESS         /* Process and thread values */
#include <os2.h>
#include <stdio.h>

#define START_PROGRAM "SIMPLE.EXE"

CHAR         szLoadError[100];
PSZ          pszArgs;
PSZ          pszEnvs;
RESULTCODES  rcReturnCodes;
ULONG        ulPid;           /* Process ID (returned)             */
ULONG        ulTarget;        /* Process ID of process to wait for */
APIRET       ulrc;            /* Return code                       */

strcpy(pszArgs,
       "-a2 -l");   /* Pass arguments "-a2" and "-l"                        */

ulTarget = 0;       /* Process ID for the most recently ended process   */

ulrc = DosExecPgm(szLoadError,           /* Object name buffer               */
                  sizeof(szLoadError),   /* Length of object name buffer     */
                  EXEC_ASYNCRESULT,      /* Asynchronous/Trace flags         */
                  pszArgs,               /* Argument string                  */
                  pszEnvs,               /* Environment string               */
                  &rcReturnCodes,        /* Termination codes                */
                  START_PROGRAM);        /* Program file name                */

if (ulrc != 0) {
    printf("DosExecPgm error: return code = %ld",
           ulrc);
    return;
}

ulrc = DosWaitChild(DCWA_PROCESS,        /* Execution options                */
                    DCWW_NOWAIT,         /* Wait options                     */
```

```
                    &rcReturnCodes,       /* Termination codes                */
                    &ulPid,               /* Process ID (returned)            */
                    ulTarget);            /* Process ID of process to wait for */

    if (ulrc != 0) {
        printf("DosWaitChild error: return code = %ld",
               ulrc);
        return;
    }
```

Creating an Exit List

You call DosExitList to add to the exit list a routine that is to be given control when a process is ended (or finishes its execution). Multiple routines can be added to the list. When the process is ending, OS/2 transfers control to each address on the list.

If there are multiple addresses on the list, each function gets control in numerical order (with 0 being first and 0FFH being last), based on a value supplied by the application when it calls DosExitList. In case of duplicate entries for this parameter, the routines will be executed in LIFO (last in, first out) order.

DosExitList requires a function code that specifies an action and a pointer to the function that is to receive control upon termination.

The following code fragment adds the locally defined function *SaveFiles* to the exit list:

```
    #define INCL_DOSPROCESS         /* Process and thread values */
    #include <os2.h>

    #define HF_STDOUT 1             /* Standard output handle    */

    VOID main(VOID)
    {
        .
        .
        .
        DosExitList(EXLST_ADD,
                    (PFNEXITLIST) SaveFiles);
        .
        .
        .
        DosExit(EXIT_PROCESS,
                0);
    }

    VOID APIENTRY SaveFiles(ULONG ulTermCode)
    {
        ULONG ulWritten;

        switch (ulTermCode) {
            case TC_EXIT:
            case TC_KILLPROCESS:
                 DosWrite(HF_STDOUT,
                          "Goodbye\r\n",
                          10,
                          &ulWritten);
                 break;

            case TC_HARDERROR:
            case TC_TRAP:
                 break;

        }
        DosExitList(EXLST_EXIT,
                    0);
    }
```

Any function that you add to the list must take one parameter. The function can carry out any task, as shown in the preceding example, but as its last action it must call DosExitList, specifying the EXLST_EXIT constant. An exit-list function must not have a return value and must not call DosExit to end.

When an exit-list routine receives control, the parameter (located at ESP+4 on the stack) contains an indicator of why the process ended. The values returned are the same as those for termination codes returned by DosWaitChild or DosExecPgm requests. These values are:

TC_EXIT (0)	Normal exit
TC_HARDERROR (1)	Hard-error halt
TC_TRAP (2)	Trap operation for a 16-bit child process
TC_KILLPROCESS (3)	Unintercepted DosKillProcess
TC_EXCEPTION (4)	Exception operation for a 32-bit child process

To execute the exit-list functions, OS/2 reassigns thread 1 after ending all other threads in the process. If thread 1 has already exited (for example, if it called DosExit without ending other threads in the process), the exit-list functions cannot be executed. In general, it is poor practice to end thread 1 without ending all other threads.

Before transferring control to the termination routines, OS/2 resets the stack to its initial value. Transfer is by way of an assembly language JMP instruction. The routine must be in the address space of the ending process. The termination routine at that address takes the necessary steps and then calls DosExitList with FunctionOrder=EXLST_EXIT. Control is then transferred to the next address in the invocation order of the exit list. When all such addresses have been processed, the process completes exiting. If a routine on the list does not call DosExitList at the completion of its processing, the process waits, and OS/2 prevents termination.

During DosExitList processing, the process is in a state of partial termination. All threads of the process are ended, except for the one executing the exit-list routines. To ensure good response to a user request to end a program, there should be minimal delay in completing termination. Termination routines should be short and fail-safe.

You can use DosExitList with the EXLST_REMOVE constant to remove a function from the list.

The designer of an exit-list routine must carefully consider which functions will be used by the routine. In general, calls to most OS/2 functions are valid in a DosExitList routine, but certain functions, such as DosCreateThread and DosExecPgm, are not.

Using Threads

A *thread* is a dispatchable unit of execution that consists of a set of instructions, related CPU register values, and a stack. Every process has at least one thread and can have many threads running at the same time. The application runs when OS/2 gives control to a thread in the process. The thread is the basic unit of execution scheduling.

Every process has at least one thread, called the *main thread* or *thread 1* . To execute different parts of an application simultaneously, you can start several threads.

A new thread inherits all the resources currently owned by the process. This means that if you opened a file before creating the thread, the file is available to the thread. Similarly, if the new thread creates or opens a resource, such as another file, that resource is available to the other threads in the process.

Creating a Thread

You use DosCreateThread to create a new thread for a process.

DosCreateThread requires the address of the code to execute and a variable to receive the identifier of the thread. The address of the code is typically the address of a function that is defined within the application.

You can pass one ULONG parameter to the thread when you start it. To pass more information to the thread, pass the address of a data structure.

You specify how you want the thread to run when you call DosCreateThread. If bit 1 of the flag parameter in the function call is 0, the thread starts immediately. If bit 1 of the flag parameter is 1, the thread starts suspended and will not run until the application calls DosResumeThread.

Each thread maintains its own stack. You specify the size of the stack when you call DosCreateThread. The amount of space needed depends on a number of factors, including the number of function calls the thread makes and the number of parameters and local variables used by each function. If you plan to call OS/2 functions, a reasonable stack size is 8192 bytes (8KB); 4096 bytes (4KB) should be the absolute minimum. If bit 1 of the flag parameter is 0, OS/2 uses the default method for initializing the thread's stack. If bit 1 of the flag parameter is 1, memory for the thread's entire stack is pre-committed.

The following code fragment creates a thread:

```
#define INCL_DOSPROCESS         /* Process and thread values */
#include <os2.h>
```

```
    #include <stdio.h>

    #define HF_STDOUT 1             /* Standard output handle    */

    VOID _System ThreadFunc(ULONG ulBeepLen);

    INT main(VOID)
      {
      ULONG ulBeepLen;
      TID   tidThread;

      ulBeepLen = 1000;
      DosCreateThread(&tidThread,       /* Thread ID returned by DosCreateThread */
                      &ThreadFunc,      /* Address of the thread function        */
                      ulBeepLen,        /* Parameter passed to thread            */
                      0,                /* Immediate execution, default stack    */
                                        /* initialization                        */
                      4096);            /* Stack size                            */

      DosWaitThread(&tidThread,
                    DCWW_WAIT);
      return 0;
      } /* end main */

/************************************************/
/* ThreadFunc                                   */
/************************************************/
    VOID _System ThreadFunc(ULONG ulBeepLen)
      {
      ULONG ulWritten; /* needed for DosWrite */

      DosBeep(750, ulBeepLen);

      DosWrite(HF_STDOUT,
               "Message from new thread\r\n",
               25,
               &ulWritten);

      DosExit(EXIT_PROCESS, 0);
      } /* end ThreadFunc */
```

A thread continues to run until it calls DosExit, returns control to OS/2, or is ended by a DosKillThread call.

In the preceding example, the thread exits when the function implicitly returns control at the end of the function.

Obtaining Information about a Thread

OS/2 creates and maintains a thread information block for each thread. An application can use DosGetInfoBlocks to access the thread information block. This function returns a pointer to a *TIB* data structure.

The code fragment below returns the address of the thread information block of the current thread. The calling thread can subsequently browse the *TIB*.

DosGetInfoBlocks also returns the address of the process information block of the current process.

Changing the Priority of a Thread

You can use DosSetPriority to change the execution priority of threads in a process. The execution priority defines when or how often a thread receives an execution time slice. Threads with higher priorities receive time slices before those with lower priorities. When a thread that is higher in priority than the currently running thread becomes ready to run, it immediately preempts the lower priority thread (the lower priority thread does not get to complete its time slice). Threads with equal priority receive time slices in a round-robin order. If you raise the priority of a thread, the thread is executed more frequently.

You can use DosSetPriority to set the priority for one thread in a process, for all threads in a process (and thus the process itself), or for threads in a child process.

A process can change the priority of any thread within itself. When a process changes the priority of threads in a descendant process,

however, only those with default priorities are changed. The priority of any thread in a descendant process that has already explicitly changed its priority from the default with DosSetPriority is not changed.

In the following code fragment, DosSetPriority lowers the priority of a process to be used as a background process:

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>

PTIB ptib;    /* thread information block          */
PPIB ppib;    /* process information block         */
APIRET rc;    /* return code from DosGetInfoBlocks */

rc = DosGetInfoBlocks(&ptib,
                      &ppib);

DosSetPriority(PRTYS_PROCESSTREE,
               PRTYC_IDLETIME,
               0,
               ppib->pib_ulpid);
```

DosGetInfoBlocks retrieves the process information blocks and thread information blocks. DosSetPriority then uses the process identifier to change the priority to idle time (idle-time processes receive the least attention by OS/2).

If you specify PRTYS_PROCESS when calling DosSetPriority, only the priority of the specified process changes. The priorities of all child processes remain unchanged.

If you specify PRTYS_THREAD in the call to DosSetPriority, you must specify a thread identifier as the last parameter. The priority of the specified thread changes, but the priorities of all other threads in the process remain unchanged.

Whenever DosSetPriority is called with a class specification, but no value is specified for priority-delta, the base priority level defaults to 0.

Suspending the Current Thread

You can temporarily suspend the execution of the current thread for a set amount of time by using DosSleep. This function suspends execution of the thread for the specified number of milliseconds. DosSleep is useful when you need to delay the execution of a task. For example, you can use DosSleep to delay a response when the user presses a DIRECTION key. The delay provides the user with enough time to observe the results and release the key.

The following code fragment uses DosSleep to suspend execution of a thread for 1000 milliseconds (1 second):

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>

DosSleep(1000);
```

Suspending and Resuming Execution of a Thread

DosSuspendThread and DosResumeThread are used to temporarily suspend the execution of a thread when it is not needed and resume execution when the thread is needed.

These functions are best used when it is necessary for a process to temporarily suspend execution of a thread that is in the middle of a task. For example, consider a thread that opens and reads files from a disk. If other threads in the process do not require input from these files, the process can suspend execution of the thread so that OS/2 does not needlessly grant control to it.

The specified thread might not be suspended immediately if it has some system resources locked that must be freed first. However, the thread is not permitted to execute further application program instructions until a corresponding DosResumeThread is called.

A thread can only suspend another thread that is within its process.

DosResumeThread is used to enable the suspended thread to resume execution.

The following code fragment temporarily suspends the execution of another thread within the same process. A subsequent call to DosResumeThread restarts the suspended thread. Assume that the thread identifier of the target thread has been placed int *ThreadID* already.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#include <os2.h>
#include <stdio.h>

TID     tidThreadID;     /* Thread identifier */
APIRET  ulrc;            /* Return code       */

ulrc = DosSuspendThread(tidThreadID);

if (ulrc != 0) {
    printf("DosSuspendThread error: return code = %ld",
           ulrc);
    return;
}

ulrc = DosResumeThread(tidThreadID);

if (ulrc != 0) {
    printf("DosResumeThread error: return code = %ld",
           ulrc);
    return;
}
```

Entering Critical Sections

A thread can prevent execution of any of the other threads in its process by calling DosEnterCritSec.

This function temporarily prevents a thread from being preempted by other threads within its process. The other threads in the process will not be executed until the current thread calls DosExitCritSec. This enables the calling thread to access a time-critical resource of the process.

The following code fragment enters a section that will not be preempted, performs a simple task, and then exits quickly.

```
#define INCL_DOSPROCESS     /* Process and thread values */
#include <os2.h>
#include <stdio.h>

BOOL    flag;         /* Program control flag */
APIRET  ulrc;         /* Return code          */

ulrc = DosEnterCritSec();

if (ulrc != 0) {
    printf("DosEnterCritSec error: return code = %ld",
           ulrc);
    return;
}

flag = TRUE;           /* Set the flag */

ulrc = DosExitCritSec();

if (ulrc != 0) {
    printf("DosExitCritSec error: return code = %ld",
           ulrc);
    return;
}
```

A count is maintained of the outstanding DosEnterCritSec requests. The count is incremented when a DosEnterCritSec request is made, and decremented when a DosExitCritSec request is made. A DosExitCritSec request will not cause normal thread dispatching to be restored while the count is greater than 0.

This count is maintained in a WORD-sized variable. If overflow occurs, the count is set to its maximum value, and an error is returned. The operation is not performed when this occurs.

Threads that call DosEnterCritSec must not must not make dynamic link calls within these critical sections. The dynamic link procedure could be using semaphores to serialize a resource. If a thread entering the critical section blocks another thread that already owns the resource which the dynamic link function is about to request, a deadlock occurs.

For example, threads of an application are serializing their access to a queue by means of a semaphore. A thread enters a critical section and makes a request to read the queue while another thread already has the semaphore that controls access to the queue. The thread that has the semaphore is now effectively blocked by DosEnterCritSec, and the thread that has requested the queue waits forever to access it.

Note: Thread 1 is the initial thread of execution. It handles all signals (Ctrl+C, Ctrl+Break, and KillProcess). If a signal occurs while DosEnterCritSec is active, thread 1 can begin execution to process the signal. Thread 1 must not access the critical resource that is being protected by the use of DosEnterCritSec.

Waiting for a Thread to End

An application might need to ensure that one thread has finished executing before another thread continues with its own task. For example, one thread might have to finish reading a disk file into memory before another thread can use the information. You can use DosWaitThread to suspend a thread until another thread has ended.

DosWaitThread places the current thread into a wait state until another thread in the current process has ended. It then returns the thread identifier of the ending thread.

The following code fragment creates three threads. The thread identifier for each thread is returned by DosCreateThread in the *atid* array. Using *&atid[0]* as a parameter in the call to DosWaitThread causes OS/2 to wait until the thread with that identifier (the thread running Thread2Func) ends.

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>

#define HF_STDOUT 1           /* Standard output handle    */

TID tidAtid[3];
ULONG ulWritten;

DosCreateThread(&tidAtid[0],
                Thread2Func,
                0,
                0,
                4096);

DosCreateThread(&tidAtid[1],
                Thread3Func,
                0,
                0,
                4096);

DosCreateThread(&tidAtid[2],
                Thread4Func,
                0,
                0,
                4096);

DosWaitThread(&tidAtid[0],
              DCWW_WAIT);

DosWrite(HF_STDOUT,
         "The thread has ended\r\n",
         27,
         &ulWritten);
```

If you set the *tid* parameter to 0 in the call to DosWaitThread, OS/2 waits only until the next thread (any thread in the process) ends. The identifier for the ended thread is then returned in the *tid* parameter.

After the threads are created as in the preceding example, the following code fragment waits until one of the threads ends, and then returns its thread identifier:

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>
```

```
    TID tid;

    tid = 0;

    DosWaitThread(&tid,
                  DCWW_WAIT);
```

The thread identifier of the next thread to end after the DosWaitThread call is returned in the *tid* parameter.

You can use DosWaitThread so that you can recover thread resources when the thread ends, or to synchronize the execution of a thread with the execution of other threads.

Ending the Current Thread

To end the execution of the current thread, call DosExit, specifying the action code as 0. It is good practice to end each thread in the application individually.

If the thread that is ending is the last thread in the process, or if the request is to end all threads in the process, then the process also ends. All threads except one are ended, and that thread executes any routines in the list specified by DosExitList. When this is complete, the resources of the process are released, and the result code that was specified in the DosExit call is passed to any thread that calls DosWaitChild for this process.

In the following code fragment, the main routine starts another program, SIMPLE.EXE, and then expects a return code of 3 to be returned. SIMPLE.EXE sets the return code with DosExit.

```
    #define INCL_DOSPROCESS           /* Process and thread values */
    #include <os2.h>
    #include <stdio.h>

    #define START_PROGRAM "SIMPLE.EXE"
    #define RETURN_OK 3

    CHAR        szLoadError[100];
    PSZ         pszArgs;
    PSZ         pszEnvs;
    RESULTCODES rcReturnCodes;
    APIRET      ulrc;

    ulrc = DosExecPgm(szLoadError,                /* Object name buffer           */
                      sizeof(szLoadError),        /* Length of object name buffer */
                      EXEC_SYNC,                  /* Asynchronous/Trace flags     */
                      pszArgs,                    /* Argument string              */
                      pszEnvs,                    /* Environment string           */
                      &rcReturnCodes,             /* Termination codes            */
                      START_PROGRAM);             /* Program file name            */

    if (ReturnCodes.codeResult == RETURN_OK)      /* Check result code            */
        printf("Things are ok...");
    else
        printf("Something is wrong...");

/*----------------SIMPLE.EXE------------------*/

    #define INCL_DOSPROCESS          /* Process and thread values */
    #include <os2.h>
    #include <stdio.h>

    #define RETURN_CODE 3

    main()
    {
        printf("Hello!\n");
        DosExit(EXIT_THREAD,         /* End thread/process */
                RETURN_CODE);        /* Result code        */
    }
```

When you specify DosExit for thread 1 (the initial thread of execution started by OS/2 for this process), all of the threads in the process are ended, and the process is ended.

Ending a Thread

DosKillThread ends a thread in the current process. DosKillThread enables a thread in a process to end any other thread in the process.

DosKillThread is used to force a thread within the current process to end without causing the entire process to be ended.

```
#define INCL_DOSPROCESS       /* Process and thread values */
#include <os2.h>

TID tidThread;    /* ThreadID of the thread to be ended */

DosCreateThread(&tidThread,
                ThreadFunction,
                0,
                0,
                4096);
.
.
.
DosKillThread(tidThread);
```

DosKillThread returns to the requestor without waiting for the target thread to complete its termination processing.

It is an invalid operation to use DosKillThread to kill the current thread.

Terminating thread 1 will cause the entire process to end similar to executing DosExit on thread 1. DosKillThread will not end a thread that is suspended. Instead the suspended thread will be ended when it resumes execution. For this reason, you should not kill the main thread of an application if there are any secondary threads that are suspended.

If the target thread is executing 16-bit code or was created by a 16-bit requester, ERROR_BUSY is returned.

Using Sessions

A session consists of at least one process and a virtual console (either full screen, or Presentation Manager window, and buffers for keyboard and mouse input). An application can manage its own child sessions by using DosStartSession, DosStopSession, DosSelectSession, and DosSetSession.

Starting a Session

DosStartSession is used to start new sessions and to specify the name of the application to be started in the new session.

There are five types of sessions that you can start: full screen, text window, Presentation Manager (PM), full screen DOS Session, and windowed DOS Session. OS/2 applications running in any of the OS/2 session types-full screen, text window, and PM-can start a session for any other application type, including DOS Sessions. Applications running in DOS Sessions cannot start sessions.

DosStartSession can be used to start either a foreground or a background session, but a new session can be started in the foreground only when the session of the caller, or one of the descendant sessions of the caller, is currently executing in the foreground.

A session can be started as an unrelated session or as a child session.

In the following code fragment, an unrelated, foreground session is created, and the application, SIMON.EXE, is started in the new session:

```
#define INCL_DOSPROCESS       /* Process and thread values */
#define INCL_DOSSESMGR
#include <os2.h>

#define HF_STDOUT 1      /* Standard output handle */
```

```
STARTDATA  sd;
PID        pidProcess;
CHAR       szBuf[CCHMAXPATH];
ULONG      ulSessionID, cbWritten;
APIRET     rc;
CHAR       szPgmName[] = "SIMON.EXE";

sd.Length = sizeof(sd);                    /* Length of the structure */
sd.Related = SSF_RELATED_INDEPENDENT;      /* Unrelated session       */
sd.FgBg = SSF_FGBG_FORE;                   /* In the foreground       */
sd.TraceOpt = SSF_TRACEOPT_NONE;           /* No tracing              */
sd.PgmTitle = (PSZ) NULL;                  /* Title is PgmName        */
sd.PgmName = szPgmName;                    /* Address of szPgmName    */
sd.PgmInputs = (PBYTE) NULL;               /* No command line args    */
sd.TermQ = (PBYTE) NULL;                   /* No terminal queue       */
sd.Environment = (PBYTE) NULL;             /* Inherits environment    */
sd.InheritOpt = SSF_INHERTOPT_PARENT;      /* Uses parent environment */
sd.SessionType = SSF_TYPE_PM;              /* PM session              */
sd.IconFile = (PSZ) NULL;                  /* Uses default icon       */
sd.PgmHandle = 0;                          /* Used by Win calls       */
sd.PgmControl = SSF_CONTROL_MAXIMIZE;      /* Starts app maximized    */
sd.InitXPos = 0;                           /* Lower left corner       */
sd.InitYPos = 0;                           /* Lower left corner       */
sd.InitXSize = 0;                          /* Ignored for maximized   */
sd.InitYSize = 0;                          /* Ignored for maximized   */
sd.ObjectBuffer = szBuf;                   /* Fail-name buffer        */
sd.ObjectBuffLen = sizeof(szBuf);          /* Buffer length           */

rc = DosStartSession(&sd, &ulSessionID, &pidProcess);

if (rc) {
    DosBeep(750,250);
    DosWrite(HF_STDOUT, "error starting new session\r\n", 28, &cbWritten);
    DosExit(EXIT_PROCESS, rc);
}
```

Before calling DosStartSession, you must create a STARTDATA data structure that defines the session to be started. Different lengths for the data structure are supported to provide compatibility and various levels of application control.

DosStartSession uses the STARTDATA structure to specify the details of the new session, such as the name of the application to start in the session, whether the new session should be started in the foreground or background, and whether the new session is unrelated or is a child session of the session calling DosStartSession.

When a session is created, the title specified in STARTDATA, (or the application title if no title is specified in STARTDATA) is added to the Window List.

The *Related* field in the STARTDATA structure specifies whether the new session is related to the session calling DosStartSession.

If the *InheritOpt* field in the STARTDATA data structure is set to 1, the new session inherits the environment and open file handles of the calling process. This applies for both unrelated and related sessions.

Controlling the Execution of Child Sessions

Once a process has started a child session, it can use DosSelectSession to control the child session.

A process calls DosSetSession to set the selectability and bonding of a child session.

Setting User Selectability of a Child Session

A process calls DosSetSession to set the *selectability* of a child session. When a child session is selectable, the user can select it from the Window List or by using Alt+Esc. When a child session is nonselectable, the user cannot select the session from the Window List or move to it by using the Alt+Esc keys.

In the following code fragment, DosSetSession makes a child session nonselectable:

```
#define INCL_DOSPROCESS       /* Process and thread values */
```

```
#define INCL_DOSSESMGR
#include <os2.h>

ULONG ulSessionID;
STATUSDATA stsdata;

stsdata.Length = sizeof(stsdata);
stsdata.SelectInd = SET_SESSION_NON_SELECTABLE;  /* Non-selectable         */
stsdata.BondInd = SET_SESSION_UNCHANGED;         /* Leaves session bonding */
                                                 /* index unchanged        */

DosSetSession(ulSessionID,
              &stsdata);
```

Once a child session is made nonselectable, the user cannot select the session from the Window List or move to it by using the Alt+Esc keys. However, the parent session can still bring the child session to the foreground by using DosSelectSession. If the session contains a Presentation Manager application or is a windowed session, the user will still be able to select it with a mouse.

The parent session can make a nonselectable child session selectable by setting the *SelectInd* field to SET_SESSION_SELECTABLE in the STATUSDATA structure. DosSetSession can be called only by a parent session and only for a child session. That is, the calling process must be the process that started the child session using DosStartSession. Neither the parent session itself nor any grandchild, nor any other descendant session beyond a child session can be the target of this call.

Additionally, DosSetSession cannot be used to change the status of a session that was started as an unrelated session. The *Related* field in the STARTDATA structure must have been set to 1 when the session was started.

Binding Child Sessions to Parent Sessions

An application can use DosSetSession to establish a bond between a parent session and one of its child sessions. When the two sessions are bound, OS/2 brings the child session to the foreground when the user selects the parent session.

In the following code fragment, a parent session is bound to the child session specified by the *ulSessionID* parameter:

```
#define INCL_DOSPROCESS       /* Process and thread values */
#define INCL_DOSSESMGR
#include <os2.h>

ULONG ulSessionID;
STATUSDATA stsdata;

stsdata.Length = sizeof(stsdata);
stsdata.SelectInd = SET_SESSION_UNCHANGED;   /* Leaves select setting alone */
stsdata.BondInd = SET_SESSION_BOND;          /* Binds parent and child      */

DosSetSession(ulSessionID, &stsdata);
```

When the application uses DosSetSession to establish a parent-child bond, any bond the parent has with another child session is broken. The application can remove the parent-child bond by calling DosSetSession with the BondInd field (in the STATUSDATA structure) set to SET_SESSION_NO_BOND.

A parent session can be executing in either the foreground or the background when it calls DosSetSession.

Switching a Session to the Foreground

An application can bring a session to the foreground, or *select* the session, by calling DosSelectSession.

DosSelectSession can only be used to select the current session or one of the current session's child sessions. It cannot be used to select a grandchild session, or any other descendant session beyond a child session, or any sessions that were started as unrelated sessions.

The session making the call, or one of its child sessions, must be executing in the foreground at the time the function is called. A process can call DosSelectSession with its own session identifier to switch itself to the foreground when one of its descendants is executing in the foreground.

The following code fragment uses DosSelectSession to switch the child session specified by the ulSessionID parameter to the foreground for five seconds. The application then switches the parent session back to the foreground:

```
#define INCL_DOSPROCESS         /* Process and thread values   */
#include <os2.h>

ULONG ulSessionID;

DosSelectSession(ulSessionID);   /* Switches child to foreground */
DosSleep(5000);                  /* Sleeps for 5 seconds         */
DosSelectSession(0);             /* Switches parent back         */
```

Terminating a Session

DosStopSession can be used by a parent session to stop one or all of its child sessions. If the child session specified in the call to DosStopSession has related sessions, the related sessions are also ended. The parent session can be running in the foreground or the background when it calls DosStopSession. If the child session is running in the foreground when it is ended, the parent session becomes the foreground session.

DosStopSession can only be called by a parent session for a child session. Neither the parent session itself, nor any grandchild, nor any other descendant session beyond a child session, nor any unrelated session, can be the target of this call.

In the following code fragment, the child session specified by the *ulSessionID* parameter is ended:

```
#define INCL_DOSPROCESS       /* Process and thread values */
#define INCL_DOSSESMGR
#include <os2.h>

ULONG ulSessionID;

DosStopSession(STOP_SESSION_SPECIFIED,
               ulSessionID);
```

An application can end all its child sessions by setting the first parameter to STOP_SESSION_ALL in the call to DosStopSession. If this is specified, the second parameter is ignored.

A process running in a child session can ignore the request to end. If the process has set up its own exception handler, it might not end immediately after the call to DosStopSession. The only way the parent process can be certain that the child session has ended is to wait for notification through the termination queue specified in the call to DosStartSession that started the session. When the child session ends, OS/2 writes a data element into the termination queue, specifying the child process identifier and the termination status.

If the process in the session specified by DosStopSession has not ended, then DosStopSession still returns a normal return code. You can ensure that a process in a session has ended by waiting for notification from the termination queue specified with DosStartSession.

Queues

Communication between processes is valuable in a multitasking operating system to enable concurrent processes to work together. Queues are one of three forms of interprocess communication (IPC), the other forms of IPC being semaphores and pipes.

This chapter describes how to create and use queues. Queues enable one or more processes to transfer data to a specific target process.

Note: The queues used for interprocess communication should not to be confused with the message queues used for communication between Presentation Manager (PM) and PM applications, nor with the printer queues used by the print spooler in managing print jobs.

The following topics are related to the information in this chapter:

- Memory (shared memory)
- Program execution and control

- Semaphores
- Pipes

About Queues

A *queue* is a named, ordered list of elements that is used to pass information between threads of the same (related) process or between different (unrelated) processes.

Processes pass information to a queue in the form of elements. An element is a 32-bit unit of information. Queue elements can be values, flags, pointers to shared memory regions, anything that can fit into 32 bits. The format of a queue element depends entirely on the process that creates the queue (the queue owner). Only the queue owner can read elements from the queue; other processes can only write to the queue. Reading an element automatically removes it from the queue.

The process that creates the queue is known as the *server process* of the queue. The other processes that access the queue are known as *client processes* .

The owner of the queue (the *server process*) can choose the order in which to read incoming information and can examine queue elements without removing them from the queue. Queue elements can be added and accessed in First-In-First-Out (FIFO), Last-In-First-Out (LIFO), or priority-based order.

Any process that has the name of a queue can open the queue and write to it. The processes writing elements to the queue must use the format determined by the queue owner.

Queues are very efficient. They pass only 32-bit sized elements, rather than large data structures. However, queues can be used only for one-way communication, because a client process can write to a queue but cannot read from one.

Typically, processes use queues to transfer information about the contents of a shared memory. The elements in the queue could contain the address and length of data areas in shared memory objects. The sending process allocates a shared memory object and gives access to the shared memory to the queue owner. The sending process can free the shared memory after writing the elements to the queue because the shared memory will not be deallocated until the queue owner frees it.

Any thread in the process that owns the queue can examine queue elements without removing them. This is called *peeking* at the queue.

OS/2 supplies the process identifier of the process that writes an element to the queue, so that a process reading from or peeking at the queue can determine the origin of the element. The process identifier is returned as part of a REQUESTDATA data structure. Threads can use the ulData field of the REQUESTDATA data structure to pass additional information about the queue element.

If the queue is empty when a process attempts to read from it, the process can either wait for an element to become available or continue executing without reading from the queue. Semaphores can be used to indicate when an element is in the queue.

Queues and Semaphores

If a process manages only one queue, it typically waits for an element to become available. However, if a process manages several queues, waiting for one queue means that other queues cannot be read. To avoid wasting time while waiting, a process can supply an event semaphore when it calls DosReadQueue or DosPeekQueue. The process can then continue to execute without actually reading an element from the queue, because DosWriteQueue will post the semaphore only when an element is ready. The semaphore remains posted until someone resets it; usually the queue owner process resets the semaphore after it reads all the available information from the queue.

If a process uses a unique semaphore for each queue, it can use DosWaitMuxWaitSem to wait for the first queue to receive an element.

Only one semaphore is permitted per queue.

Queue Servers and Clients

The server process and its threads have certain queue-managing privileges. Only the server process and its threads can:

- Examine queue elements without removing them (DosPeekQueue)
- Remove elements from the queue (DosReadQueue)
- Purge all the elements in a queue (DosPurgeQueue)
- Write to the queue without opening it first (DosWriteQueue)

- Delete the queue (DosCloseQueue).

Both server and client processes can query the number of elements in the queue using DosQueryQueue.

Client processes can query the queue (DosQueryQueue) and add elements to it (DosWriteQueue), but they must first gain access to the queue by calling DosOpenQueue. When a client process is finished with a queue, it ends its access to the queue by calling DosCloseQueue. (Note that, unlike the server process and its threads, a client process cannot use DosCloseQueue to delete a queue.)

When a queue is opened by a client process, an access count is set to 1. Each client process has its own access count. The access count is incremented whenever a thread in a process opens the queue and decremented whenever a thread in the process closes the queue. Access to the queue by the client process ends when the access count for the process reaches 0. When the server process closes the queue, the queue is terminated and removed from the system.

Queue Element Order

DosReadQueue reads either a specified element or the first element in the queue. The first element in the queue depends on the queue type, which is specified when the queue is created. A queue can have FIFO, LIFO, or priority ordering.

Priority values range from 0 (lowest priority) through 15 (highest priority). The writing process assigns a priority to a queue element when the element is written to the queue. DosReadQueue reads elements from the queue in descending order of priority, regardless of the order in which DosWriteQueue placed the elements in the queue. Elements with equal priority are read in FIFO order.

Obtaining Information about Queues and Queue Elements

Any thread in the process that owns the queue can use DosPeekQueue to examine the elements in the queue to determine which one to actually read. Each call to DosPeekQueue returns the identifier of the next element in the queue, so the function can be called repeatedly to move through the queue. The identifier of the desired element can then be supplied to DosReadQueue.

Any process that has opened a queue can use DosQueryQueue to determine the number of elements in the queue. This function also returns an error value if the queue owner has closed the queue.

Using Queues

Queues are useful for a process to manage input from other processes. The examples in the following sections show how to create and use queues.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Creating a Queue

A thread creates a queue by using DosCreateQueue and specifying a queue name and the queue type as arguments. The queue name must be unique and have the following form:

```
\QUEUES\QueName
```

The "\QUEUES\" is required, though it need not be uppercase. It is not a subdirectory.

The *QueName* parameter must conform to the rules for OS/2 file names, although no actual file is created for the queue.

The process that creates the queue is known as the *server process* of the queue. The other processes that access the queue are known as *client processes* .

The following code fragment creates a FIFO queue named \queues\sample.que:

```
#define INCL_DOSQUEUES   /* Queue values */
#include <os2.h>

HQUEUE hq;

DosCreateQueue(&hq, QUE_FIFO |
               QUE_CONVERT_ADDRESS,
               "\\queues\\sample.que");
```

When the server process creates the queue, it determines whether the ordering of queue elements is based on arrival (FIFO or LIFO) or priority. If the ordering is based on priority, then priority values must be assigned whenever data is added to the queue.

The server must also specify whether OS/2 is to convert 16-bit addresses of elements placed in the queue by 16-bit processes to 32-bit addresses.

After a process has created a queue, any thread in that process can access the queue with equal authority.

Allocating Memory for Queue Data

When queues are used only to pass the addresses to data rather than the data itself, processes must allocate shared memory objects for storing queue data. The two most common methods of storing queue data are:

- Using a named shared memory object - for related processes
- Using unnamed shared memory objects - for unrelated processes

Named Shared Memory Objects

Related processes generally use a single named shared memory object for storing queue data. The server process allocates the memory object by calling DosAllocSharedMem. Care must be taken to ensure that the memory object is large enough to meet application requirements.

The name of the shared memory object is established by agreement among the server and the client processes. For simplicity, the name can be the same as the queue name, except that the prefix \SHAREMEM\ must be used instead of \QUEUES\.

A client process accesses the named shared memory object by calling DosGetNamedSharedMem. It must then call DosOpenQueue to gain access to the queue of the server.

Before the server process ends, it releases the memory object by calling DosFreeMem.

Unnamed Shared Memory Objects

Unrelated processes generally use unnamed shared memory objects for storing queue data. This makes it possible for a client process to store data in a shared memory object without knowing its name. To use unnamed shared memory objects for storing queue data, the server process must take the following steps whenever it is called by a client:

1. Save the process identification (PID) of the client process.
2. Allocate an unnamed shared memory object for the data of the client process by calling DosAllocSharedMem.
3. Give the client process the capability of accessing the memory object by calling DosGiveSharedMem with the client's PID.

The client process must then call DosOpenQueue to gain access to the server's queue.

After each client completes its queue requests or ends, the server calls DosFreeMem to release the client's memory object.

Opening a Queue

Once the queue is created, the server process of the queue and the threads of the server process have immediate access to the queue and can proceed to access the queue. A client process must request access to a queue by calling DosOpenQueue. Once the queue is open, the client process can add an element to the queue with DosWriteQueue, and it can query the number of elements in the queue with DosQueryQueue.

DosOpenQueue retrieves the queue handle and the process identifier of the queue owner. The function also increments the queue's access count.

The following code fragment shows how another process would open the queue created with DosCreateQueue.

```
#define INCL_DOSQUEUES
#include<os2.h>

#define HF_STDOUT 1    /* Standard output handle */

HQUEUE  hq;
PID     pidOwner;
ULONG   ulWritten;
APIRET  ulrc;

ulrc = DosOpenQueue(&pidOwner,
                    &hq,
                    "\\queues\\sample.que");

if (ulrc) {
    DosWrite(HF_STDOUT,
             "\r\n Queue open failed. \r\n",
             24,
             &ulWritten);

    DosExit(EXIT_PROCESS,
            1);
}
else {
    DosWrite(HF_STDOUT,
             "\r\n Queue opened. \r\n",
             19,
             &ulWritten);
}
```

When it is finished with the queue, a thread in the client process ends its access by calling DosCloseQueue. DosCloseQueue decrements the access count for the process each time it is called. When the access count reaches 0, the connection between the client process and the queue is terminated.

After a process has opened a queue, any thread in that process can access the queue with equal authority.

Note: If a queue was created by a call to the 16-bit DosCreateQueue, then it is not accessible to 32-bit DosOpenQueue requests, and ERROR_QUE_PROC_NO_ACCESS will be returned.

Writing to a Queue

The server process and any of its threads can add an element to a queue simply by calling DosWriteQueue. A client process, however, must first request access to the queue by calling DosOpenQueue.

Processes that communicate by passing the addresses of shared memory objects through the queue must have a shared memory object that they each have access to. Once a process opens the queue, it can allocate shared memory by using DosAllocMem with the OBJ_GIVEABLE attribute and then give the shared memory to the queue owner with DosGiveSharedMem.

A process that has opened a queue can write to the queue by using DosWriteQueue. The writing process must create elements in a form that the queue owner can read.

The following code fragment adds an element to a queue. Assume that the caller has placed the handle of the queue into *QueueHandle*

already. Assume also that *DataBuffer* has been set to point to a data element in shared memory, and that *DataLength* has been set to contain the length of the data element in shared memory.

```
#define INCL_DOSQUEUES   /* Queue values */
#include <os2.h>
#include <stdio.h>

HQUEUE   hqQueueHandle;   /* Queue handle                          */
ULONG    ulRequest;       /* Request-identification data           */
ULONG    ulDataLength;    /* Length of element being added         */
PVOID    pDataBuffer;     /* Element being added                   */
ULONG    ulElemPriority;  /* Priority of element being added       */
APIRET   ulrc;            /* Return code                           */

ulRequest = 0;            /* Assume that no special data is being */
                          /* sent along with this write request   */

ulElemPriority = 0;       /* For priority-based queues: add the   */
                          /* new queue element at the logical end */
                          /* of the queue                         */

ulrc = DosWriteQueue(hqQueueHandle,
                     ulRequest,
                     ulDataLength,
                     pDataBuffer,
                     ulElemPriority);

if (ulrc != 0) {
    printf("DosWriteQueue error: return code = %ld",
           ulrc);
    return;
}
```

Once the process has written to the queue, it frees the shared memory. However, the memory will not be freed until the queue owner also frees it.

If the queue was created as a priority-based queue (as specified in the *QueueFlags* parameter of DosCreateQueue), then the priority of the element that is being added must be specified.

If the server process has ended, or if it has closed the queue before DosWriteQueue is called, then ERROR_QUE_INVALID_HANDLE is returned.

Reading from a Queue

The queue owner (server process) and its threads can read an element from the queue by using DosReadQueue. The owner can read the first element in the queue by specifying 0 as the element number. Alternatively, the owner can read a particular element in the queue by specifying an element code returned from DosPeekQueue. This function is not available to client processes.

DosReadQueue can either remove queue elements in the order that was specified when the queue was created (FIFO, LIFO, or priority), or it can use an element identifier from DosPeekQueue as input to remove a previously examined element.

The following code fragment reads an element from the queue. Assume that the caller has placed the handle of the queue into *QueueHandle* already and that the identifier of the process that owns the queue has been placed into *OwningPID* already.

```
#define INCL_DOSQUEUES   /* Queue values */
#include <os2.h>
#include <stdio.h>

HQUEUE        hqQueueHandle;   /* Queue handle                */
REQUESTDATA   rqRequest;       /* Request-identification data */
ULONG         ulDataLength;    /* Length of element received  */
PULONG        pulDataAddress;  /* Address of element received */
ULONG         ulElementCode;   /* Request a particular element */
BOOL32        bNoWait;         /* No wait if queue is empty   */
BYTE          bElemPriority;   /* Priority of element received */
HEV           hevSemHandle;    /* Semaphore handle            */
PID           pidOwningPID;    /* PID of queue owner          */
APIRET        ulrc;            /* Return code                 */

rqRequest.pid = pidOwningPID;     /* Set request data block to indicate  */
                                  /* queue owner                         */
```

```
ulElementCode = 0;              /* Indicate that the read should start */
                                /* at the front of the queue           */

bNoWait = 0;                    /* Indicate that the read should wait  */
                                /* if the queue is currently empty     */

hevSemHandle = 0;               /* Unused since this is a call that    */
                                /* waits synchronously                 */

ulrc = DosReadQueue(hqQueueHandle,
                    &rqRequest,
                    &ulDataLength,
                    (PVOID *) &pulDataAddress,
                    ulElementCode,
                    bNoWait,
                    &bElemPriority,
                    hevSemHandle);

if (ulrc != 0) {
    printf("DosReadQueue error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, *DataLength* contains the size of the element on the queue that is pointed to by the pointer within *DataAddress* , *ElemPriority* has been updated to contain the priority of the queue element pointed to by *DataAddress* , and *Request.ulData* contains any special data that the DosWriteQueue caller placed into the queue.

If the queue is empty and *NoWait* is set to DCWW_WAIT (0), the calling thread waits until an element is placed in the queue. If the queue is empty and *NoWait* is set to DCWW_NOWAIT (1), DosReadQueue returns immediately with ERROR_QUE_EMPTY.

If *NoWait* is set to DCWW_NOWAIT, an event semaphore must be provided so that the calling thread can determine when an element has been placed in the queue. The semaphore is created by calling DosCreateEventSem, and its handle is supplied as a DosReadQueue parameter. The first time an event semaphore handle is supplied in a DosReadQueue or DosPeekQueue request for which DCWW_NOWAIT has been specified for a particular queue, the handle is saved by the system. The same handle must be supplied in all subsequent DosReadQueue and DosPeekQueue requests that are called for that queue; if a different handle is supplied, ERROR_INVALID_PARAMETER is returned.

When a client process adds an element to the queue, the system automatically opens the semaphore (if necessary) and posts it. The server can either call DosQueryEventSem periodically to determine whether the semaphore has been posted, or it can call DosWaitEventSem. DosWaitEventSem causes the calling thread to block until the semaphore is posted.

After the event semaphore has been posted, the calling thread must call DosReadQueue again to remove the newly added queue element.

If QUE_CONVERT_ADDRESS is specified in the call to DosCreateQueue, OS/2 will automatically convert 16-bit addresses to 32-bit addresses.

Peeking at a Queue

The server process and its threads can examine a queue element by calling DosPeekQueue. This function is not available to client processes.

Unlike DosReadQueue, DosPeekQueue does not remove the element from the queue.

DosPeekQueue can either examine elements in the order that was specified when the queue was created (FIFO, LIFO, or priority), or it can examine the next element in the queue after a previous DosPeekQueue request has been called. By making multiple DosPeekQueue requests, the server process can search through a queue, examining each element in turn. When it locates the element it is searching for, the server process can remove the element from the queue by calling DosReadQueue.

If several threads are using the same queue, the process writing to the queue can use the ulData field of the REQUESTDATA data structure to indicate that an element is directed to a particular thread. The thread can peek at the queue whenever data is available and read any elements containing the appropriate value in the ulData field.

The following code fragment shows how a thread can use DosPeekQueue to examine the elements in a queue. Assume that a previous call to DosOpenQueue provided the queue handle that is contained in *QueueHandle* . Assume that the identifier of the process that owns the queue has been placed into *OwningPID* already.

```
#define INCL_DOSQUEUES       /* Queue values */
#define INCL_DOSPROCESS      /* needed for DCWW_WAIT */
```

```
#include <os2.h>
#include <stdio.h>

HQUEUE        hqQueueHandle;   /* Queue handle                  */
REQUESTDATA   rqRequest;       /* Request-identification data   */
ULONG         ulDataLength;    /* Length of examined element    */
PVOID         pDataAddress;    /* Address of examined element   */
ULONG         ulElementCode;   /* Indicator of examined element */
BOOL32        bNoWait;         /* No wait if queue is empty     */
BYTE          bElemPriority;   /* Priority of examined element  */
HEV           hevSemHandle;    /* Semaphore handle              */
PID           pidOwningPID;    /* PID of queue owner            */
APIRET        ulrc;            /* Return code                   */

rqRequest.pid = pidOwningPID; /* Set request data block to indicate   */
                              /* queue owner                          */

ulElementCode = 0;           /* Indicate that the peek should start  */
                             /* at the front of the queue            */

bNoWait = DCWW_WAIT;          /* Indicate that the peek call should   */
                              /* wait if the queue is currently empty */

hevSemHandle = 0;             /* Unused since this is a call that     */
                              /* synchronously waits                  */

ulrc = DosPeekQueue(hqQueueHandle,
                    &rqRequest,
                    &ulDataLength,
                    &pDataAddress,
                    &ulElementCode,
                    bNoWait,
                    &bElemPriority,
                    hevSemHandle);

if (ulrc != 0) {
    printf("DosPeekQueue error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, *DataLength* contains the size of the element on the queue that is pointed to by the pointer within *DataAddress*, *ElementCode* has been updated to indicate the next queue element, *ElemPriority* has been updated to contain the priority of the queue element pointed to by *DataAddress*, and *Request.ulData* contains any special data that the DosWriteQueue caller placed into the queue.

If the queue is empty and *NoWait* is set to DCWW_WAIT (0), the calling thread waits until an element is placed in the queue. If the queue is empty and *NoWait* is set to DCWW_NOWAIT (1), DosPeekQueue returns immediately with ERROR_QUE_EMPTY.

If *NoWait* is set to DCWW_NOWAIT, an event semaphore must be provided so that the calling thread can determine when an element has been placed in the queue. The semaphore is created by calling DosCreateEventSem, and its handle is supplied as a DosPeekQueue parameter. The first time an event semaphore handle is supplied in a DosPeekQueue or DosReadQueue request for which DCWW_NOWAIT has been specified for a particular queue, the handle is saved by the system. The same handle must be supplied in all subsequent DosPeekQueue and DosReadQueue requests that are called for that queue; if a different handle is supplied, ERROR_INVALID_PARAMETER is returned.

When a client process adds an element to the queue, the system automatically opens the semaphore (if necessary) and posts it. The server can either call DosQueryEventSem periodically to determine whether the semaphore has been posted, or it can call DosWaitEventSem. DosWaitEventSem causes the calling thread to block until the semaphore is posted.

After the event semaphore has been posted, the calling thread must call DosPeekQueue again to examine the newly added queue element.

If QUE_CONVERT_ADDRESS is specified in the call to DosCreateQueue, OS/2 will automatically convert 16-bit addresses to 32-bit addresses.

Purging a Queue

The server process or any of its threads can empty a queue of all its elements by calling DosPurgeQueue. This function is not available to client processes.

Warning: This is an unconditional purge of all elements in the queue.

The following code fragment shows how the owner of a queue can empty the queue of all data elements. Assume that the owner of the queue has saved the queue's handle (obtained in a previous call to DosCreateQueue) in *QueueHandle* .

```
#define INCL_DOSQUEUES    /* Queue values */
#include <os2.h>
#include <stdio.h>

HQUEUE   hqQueueHandle;     /* Queue handle */
APIRET   ulrc;              /* Return code  */

ulrc = DosPurgeQueue(hqQueueHandle);

if (ulrc != 0) {
    printf("DosPurgeQueue error: return code = %ld",
           ulrc);
    return;
}
```

Closing a Queue

DosCloseQueue can be used both to end access to a queue and to delete a queue. The action taken as a result of a DosCloseQueue request depends on:

- Whether the requesting process is the server process or a client process
- The value of the access count, which is maintained for each client process by OS/2. The access count for a client process is incremented whenever DosOpenQueue is called, and decremented whenever DosCloseQueue is called.

If the requesting process is a client, and the access count equals 0, DosCloseQueue ends the client's access to the queue, but the queue itself is not affected. If the access count does not equal 0, the count is decremented, but the process retains access to the queue.

If the requesting process is the server, DosCloseQueue purges any outstanding elements from the queue and deletes the queue regardless of the access count; client processes that still have the queue open receive ERROR_QUE_INVALID_HANDLE on their next request.

Semaphores

Communication between processes is valuable in a multitasking operating system to enable concurrent processes to work together. Semaphores are one of three forms of interprocess communication (IPC), the other forms of IPC being pipes and queues.

This chapter describes how to create and use semaphores. Semaphores enable an application to signal the completion of tasks and control access to resources that are shared between more than one thread or process.

The following topics are related to the information in this chapter:

- Memory (shared memory)
- Program execution and control
- Pipes
- Queues

About Semaphores

Semaphores signal the beginning or ending of an operation and provide mutually exclusive ownership of resources. Typically, semaphores are used to prevent more than one process or thread within a process from accessing a resource, such as shared memory, at the same time.

Semaphores are defined by OS/2 and reside in an internal memory buffer. They are divided into three types, according to the functionality they provide:

- *Event semaphores* enable a thread to notify waiting threads that an event has occurred. The waiting threads then resume execution, performing operations that are dependent on the completion of the signaled event.
- *Mutual exclusion (mutex) semaphores* enable threads to serialize their access to shared resources. That is, ownership of a mutex semaphore is used by cooperating threads as a prerequisite for performing operations on a resource. (Threads cooperate by using the mutex semaphore functions to ensure that access to the resource is mutually exclusive.)
- *Multiple wait (muxwait) semaphores* enable threads to wait either for multiple events to occur, or for multiple resources to become available. Alternatively, a flag can be set so that a thread waits for any one of multiple events to occur, or for any one of multiple resources to become available.

Event Semaphores

An event semaphore provides a signaling mechanism among threads or processes, ensuring that events occur in the desired sequence. Event semaphores are used by one thread to signal other threads that an event has occurred. An application can use this type of semaphore to block a thread or process until the event has occurred.

An event semaphore has two states, reset and posted. When an event semaphore is in the reset state, OS/2 blocks any thread or process that is waiting on the semaphore. When an event semaphore is in the posted state, all threads or processes waiting on the semaphore resume execution.

For example, assume thread 1 is allocating a shared memory object and threads 2 and 3 must wait for the memory to be allocated before they attempt to examine its contents. Before thread 1 allocates the memory, it creates an event semaphore, specifying the initial state of the semaphore as reset. (If the event semaphore has already been created, thread 1 simply resets the semaphore.) Threads 2 and 3 use DosWaitEventSem to wait for the semaphore to signal that the event, in this case the allocation and preparation of the shared memory object, has been completed. Because the semaphore was reset by thread 1, threads 2 and 3 are blocked when they call DosWaitEventSem. After thread 1 has finished allocating and placing data in the shared memory object, it signals the completion of its task by posting the event semaphore. The posting of the event semaphore unblocks threads 2 and 3, enabling them to resume execution. They can then proceed to examine the contents of the allocated memory.

In the example above, one thread controls the resetting and posting of the event semaphore, while other threads merely wait on the semaphore. Another approach could be for an application or thread to reset an event semaphore, then block itself on that semaphore. At a later time, another application or thread would post the event semaphore, unblocking the first thread.

Mutual Exclusion (Mutex) Semaphores

A mutual exclusion (mutex) semaphore protects resources (such as files, data in memory, and peripheral devices) from simultaneous access by several processes. Mutex semaphores enable threads to serialize their access to resources. It does so by preventing the processes from concurrently executing the sections of code through which access is made. These sections of code are called critical sections. For example, a mutex semaphore could be used to prevent two or more threads from simultaneously writing to the same file on a disk.

Before a thread can execute a mutex-protected critical section, it must request and receive ownership of the mutex semaphore. Only the thread that has gained ownership of the mutex semaphore is permitted to perform operations on the protected resource. Only one thread at a time can own the mutex semaphore, and the owner thread retains ownership until it finishes executing its critical section. When finished, the owner thread releases the mutex semaphore, enabling another thread to become the owner.

When a thread requests ownership of a mutex semaphore that is already owned, OS/2 blocks the thread. When more than one thread requests ownership of the same semaphore, OS/2 queues the requests and grants subsequent ownership based on the thread's priority and the order in which the requests were received.

If more than one thread is blocked on a DosRequestMutexSem request, then ownership is given to the thread that has the highest priority level. If more than one of the waiting threads have the same priority level, then FIFO ordering is used to determine which thread is unblocked and given ownership of the semaphore.

For example, both thread 1 and thread 2 must write information to the same disk file. Thread 1 claims ownership of an agreed-upon mutex semaphore and starts writing its information to the file. If thread 2 also requests ownership of the semaphore, it will be blocked. When thread 1 has finished writing to the file, it releases the semaphore. OS/2 then unblocks thread 2 and designates it as the new owner of the semaphore so that it can write to the file.

During process termination, after delivery of process-termination exceptions and unwind exceptions, if any threads in the process aside from Thread 1 (the main thread) own a mutex semaphore, ownership of the semaphore (and therefore, the shared resource) passes to Thread 1. This gives Thread 1 a last chance to clean up the semaphore and the shared resource before the process ends. If Thread 1 ends without releasing the semaphore, all threads that are currently waiting on ownership of the semaphore will be unblocked with the SEM_OWNER_DIED return code. Any thread that attempts to open it or request ownership of the semaphore will receive a SEM_OWNER_DIED return code.

The recommended way to clean up semaphores, and other resources, is for each thread, especially Thread 1, to have an exception handler to handle clean up during process termination (the XCPT_PROCESS_TERMINATE or XCPT_ASYNC_PROCESS_TERMINATE exceptions). When it is not possible to register an exception handler for a thread, (a DLL, for example, must de-register its exception handlers when it returns control to the thread that called it), you should add a clean up routine to the exit list of the process.

Multiple Wait (Muxwait) Semaphores

A multiple wait (muxwait) semaphore enables a thread to wait on several event or mutex semaphores simultaneously. A muxwait semaphore is a compound semaphore that consists of a list of up to 64 event semaphores or mutex semaphores (the two types cannot be mixed).

A flag is set when the muxwait semaphore is created to enable threads to use the semaphore in either of two ways:

- Threads can wait for all of the mutex semaphores to be released, or for all of the event semaphores to be posted.
- Threads can wait for any one of the mutex semaphores in the list to be released, or for any one of the event semaphores in the list to be posted.

Depending on the value of the flag, a muxwait semaphore is said to have cleared when either any or all of the semaphores in the muxwait list have been posted or released.

For example, suppose a thread requires access to several regions of shared memory at the same time. OS/2 blocks the thread until the thread acquires ownership of all the mutex semaphores protecting the shared regions. The thread can then access all the memory regions. Meanwhile, OS/2 prevents access by other threads.

Named and Anonymous Semaphores

A semaphore can be either named or anonymous. A named semaphore is always shared; that is, it is always available to any process that knows the name. An anonymous semaphore can be either private to a process or shared among processes, depending on whether the application includes the DC_SEM_SHARED flag in the function that creates the semaphore. A semaphore intended for use solely among threads of the same process can be anonymous and private.

OS/2 creates a named semaphore when an application specifies a name in the function that creates the semaphore. The name must have the following form:

```
\SEM32\SemName
```

The "\SEM32\" is required, though it need not be uppercase. The semaphore name must conform to the rules for OS/2 file names, although no actual file is created for the semaphore. If the application does not specify a name in the function that creates the semaphore, OS/2 creates an anonymous semaphore.

OS/2 permits a system-wide maximum of 65536 (64K) shared semaphores. In addition, each process can use up to 65536 (64K) private semaphores.

A shared muxwait semaphore must contain either all shared event semaphores or all shared mutex semaphores. However, a private muxwait semaphore can contain a combination of shared and private event or mutex semaphores. OS/2 generates a unique handle when it creates a semaphore. Processes must obtain this handle before they can access the semaphore. A semaphore's handle is always available to the process that created the semaphore. A process can obtain the handle of a named semaphore created in another process by using the appropriate semaphore-opening function. A process that requires access to an anonymous shared semaphore that was created in another process must obtain the handle of the semaphore through some other form of interprocess communication, such as a pipe or a queue.

Semaphore Management

After one process creates a semaphore, threads in other processes must open the semaphore before they can access it. (Creating a semaphore automatically opens it for the creating process.) The open operation ensures that the process is a valid user of the semaphore. OS/2 keeps track of the number of open operations that each process performs on a semaphore. A process can have up to 65535 (64K - 1) open operations performed on a semaphore at any one time.

If a process finishes using a semaphore and will not use it again, the process should close the semaphore so that OS/2 can free the memory the semaphore is using. OS/2 returns the ERROR_SEM_BUSY error value if a thread tries to close a semaphore that has another thread in

the same process still waiting for it.

If a process terminates with open semaphores, OS/2 automatically closes the semaphores for that process.

Semaphores reside in a memory buffer rather than a disk file. Therefore, when the last process that has a semaphore open exits or closes that semaphore, OS/2 frees the associated handle or name.

When an application calls a function that causes a thread to wait on a semaphore, the application can specify the amount of time for the thread to wait. When the interval elapses without the semaphore being posted or released, the function returns the ERROR_TIMEOUT error value and the thread continues running. The application can provide a specific time-out value in milliseconds, or it can specify either the SEM_INDEFINITE_WAIT or the SEM_IMMEDIATE_RETURN flag. If a thread is interrupted while it is waiting on a semaphore, the ERROR_INTERRUPT error value is returned to the caller.

Using Event Semaphores

An application can use an event semaphore to trigger execution of other processes. This is useful if, for example, one process provides data to many other processes. Using an event semaphore frees the other process from the trouble of polling to determine when new data is available.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Creating an Event Semaphore

Processes create an event semaphore by using DosCreateEventSem. The process that controls the event or resource is usually the one that creates the semaphore, but it does not have to be.

Threads in the process that creates the semaphore do not have to open the semaphore before using it. DosCreateEventSem obtains access to the semaphore for the calling process and its threads. Threads in other processes must call DosOpenEventSem to open the semaphore before they can use it.

Event semaphores can be defined as either private or shared:

- Private semaphores are always unnamed and are therefore always identified by their handles. They can be used only by threads within a single process.
- Shared semaphores can be either named or unnamed. If named, they can be opened using either the name or the handle. The handle returned by DosOpenEventSem is then used to identify the semaphore for all other functions. Semaphore names must include the prefix \SEM32\ and must conform to file system naming conventions. Shared semaphores can be used by threads in multiple processes.

In the following code fragment, the controlling process creates a named event semaphore and posts the semaphore after writing data to a shared file:

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>

HEV hevWriteEvent;

DosCreateEventSem("\\sem32\\wrtevent",          /* Named-shared semaphore */
                  &hevWriteEvent, 0, FALSE);   /* Initially reset        */
    .
    .
    /* Write data to shared file. */
    .
    .
DosPostEventSem(hevWriteEvent);                /* Posts the event        */
    .
    .    /* Continue execution. */
    .
```

There is a system-wide limit of 65536 (64K) shared semaphores (including mutex, event, and muxwait semaphores); in addition, each process

can have up to 65536 (64K) private semaphores.

When an event semaphore is created, a flag is used to specify the initial state of the event semaphore, either reset or posted. If the initial state is reset, a thread that calls DosWaitEventSem will be blocked until a process that has access to the semaphore uses DosPostEventSem to post the event semaphore. If the initial state is posted, then a thread that calls DosWaitEventSem will return immediately to continue its execution. If the thread calling DosWaitEventSem is not in the process that created the semaphore, the thread must open the semaphore with DosOpenEventSem before calling DosWaitEventSem.

OS/2 maintains a usage count for each semaphore. DosCreateEventSem initializes the usage count to 1. Thereafter, each call to DosOpenEventSem increments the count, and each call to DosCloseEventSem decrements it.

Opening an Event Semaphore

When a process creates an event semaphore, all of the threads that belong to the process have immediate access to the semaphore.

Threads in other processes must open the semaphore by calling DosOpenEventSem before they can use the semaphore in any other event semaphore function.

The following code fragment shows how processes can open an event semaphore that was created in a different process and then wait for the event to be posted:

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>

HEV hevEventHandle = 0;                /* Must be 0 because we are opening */
                                       /* the semaphore by name            */

DosOpenEventSem("\\sem32\\wrtevent", &hevEventHandle);

DosWaitEventSem(hevEventHandle,
                SEM_INDEFINITE_WAIT);   /* Waits until event is posted      */
    .
    .    /* Read from file when event is posted. */
    .
```

Applications can open an event semaphore by name or by handle. If the name is used to open the semaphore, as in the code fragment above, the handle parameter must be 0. If the handle is used to open the semaphore, the name parameter must be NULL.

Access to semaphores is on a per-process basis. Therefore, a semaphore that has been opened by one thread in a process is open to all other threads in that process as well.

DosOpenEventSem merely provides access to an event semaphore. In order to wait for an event semaphore to be posted, a thread must call DosWaitEventSem. In order to post or reset an open event semaphore, a thread uses DosPostEventSem or DosResetEventSem respectively.

When a process no longer requires access to an event semaphore, it closes the semaphore by calling DosCloseEventSem. If a process ends without closing an open semaphore, the semaphore is closed by OS/2.

Each call to DosOpenEventSem increments the usage count of the semaphore. This count is initialized to 1 when the semaphore is created and is decremented by each call to DosCloseEventSem. When the usage count reaches 0, the semaphore is deleted by OS/2.

Calls to DosOpenEventSem and DosCloseEventSem can be nested, but the usage count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_OPENS is returned.

Closing an Event Semaphore

When a process no longer requires access to an event semaphore, it closes the semaphore by calling DosCloseEventSem.

The following code fragment closes an event semaphore. Assume that the handle of the semaphore has been placed into *HEV* already.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HEV     hev;     /* Event semaphore handle */
```

```
APIRET  ulrc;    /* Return code           */

ulrc = DosCloseEventSem(hev);

if (ulrc != 0) {
    printf("DosCloseEventSem error: return code = %ld",
           ulrc);
    return;
}
```

Calls to DosOpenEventSem and DosCloseEventSem can be nested, but the usage count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_OPENS is returned.

If a process ends without closing an open semaphore, the semaphore is closed by OS/2.

Each call to DosCloseEventSem decrements the usage count of the semaphore. This count is initialized to 1 when the semaphore is created and is incremented by each call to DosOpenEventSem. When the usage count reaches 0, the semaphore is deleted from OS/2. The call to DosCloseEventSem that decrements the usage count to 0 and causes the semaphore to be deleted is referred to as the *final close* . If a thread attempts to perform the final close for a semaphore while another thread in the same process is still waiting for it, ERROR_SEM_BUSY is returned.

Resetting an Event Semaphore

DosResetEventSem resets an event semaphore if it is not already reset, and returns the number of times the semaphore was posted since it was last reset. All threads that subsequently call DosWaitEventSem for this semaphore will be blocked.

Any thread belonging to the process that created the event semaphore can change the state of the semaphore to reset by calling DosResetEventSem. Threads in other processes can also call DosResetEventSem, but they must first gain access to the semaphore by calling DosOpenEventSem.

When an event semaphore is in the reset state, any thread that calls DosWaitEventSem to wait for the semaphore will be blocked. When the event semaphore is posted, all of the threads that are waiting for the semaphore are released to continue execution.

The following code fragment resets an event semaphore. Assume that the handle of the semaphore has been placed into *HEV* already.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HEV     hev;        /* Event semaphore handle                       */
ULONG   ulPostCt;   /* Post count for the event semaphore (returned) */
APIRET  ulrc;       /* Return code                                  */

ulrc = DosResetEventSem(hev, &ulPostCt);

if (ulrc != 0) {
    printf("DosResetEventSem error: return code = %ld",
           ulrc);
    return;
}
```

DosResetEventSem returns the post count of the event semaphore and resets the post count to 0. The post count is the number of times the semaphore has been posted (using DosPostEventSem) since the last time the semaphore was in the reset state. (An event semaphore can be reset when it is created, as well as by calling DosResetEventSem.) The post count can also be obtained by calling DosQueryEventSem.

If the event semaphore is already reset when DosResetEventSem is called, ERROR_ALREADY_RESET is returned, along with a post count of 0. The semaphore is not reset a second time.

Posting an Event Semaphore

DosPostEventSem posts the semaphore, if it is not already posted, and increments the post count. All threads that have called DosWaitEventSem for this semaphore are unblocked and resume execution. Threads that call DosWaitEventSem after the event semaphore has been posted and before the next time it is reset, will return immediately from a call to DosWaitEventSem and continue execution. If the

semaphore is subsequently reset, threads that call DosWaitEventSem will again be blocked.

Any thread in the process that created an event semaphore can post the semaphore by calling DosPostEventSem. Threads in other processes can also call DosPostEventSem, but they must first gain access to the semaphore by calling DosOpenEventSem.

The following code fragment posts a system event semaphore. Assume that the handle of the semaphore has been placed into *HEV* already.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HEV     hev;     /* Event semaphore handle */
APIRET  ulrc;    /* Return code            */

ulrc = DosPostEventSem(hev);

if (ulrc != 0) {
    printf("DosPostEventSem error: return code = %ld",
           ulrc);
    return;
}
```

OS/2 maintains a post count for each event semaphore. The post count is the number of times the semaphore has been posted (with DosPostEventSem) since the last time the semaphore was in the reset state.

If the event semaphore is reset when DosPostEventSem is called, the semaphore is posted and the post count is set to 1. If the event semaphore is already posted when DosPostEventSem is called, the post count is incremented, and ERROR_ALREADY_POSTED is returned to the calling thread.

The post count is returned as output by DosResetEventSem; it can also be obtained by calling DosQueryEventSem.

The maximum number of times an event semaphore can be posted is 65535. The value of the post count cannot exceed 65535. If an attempt is made to exceed this number, DosPostEventSem returns ERROR_TOO_MANY_POSTS.

Waiting for an Event Semaphore

Any thread in the process that created an event semaphore can wait for the semaphore to be posted by calling DosWaitEventSem. Threads in other processes can also call DosWaitEventSem, but they must first gain access to the semaphore by calling DosOpenEventSem.

If the semaphore is already posted when DosWaitEventSem is called, the function returns immediately, and the thread continues to run. Otherwise, the thread is blocked until the semaphore is posted.

The following code fragment causes the calling thread to wait until the specified event semaphore is posted. Assume that the handle of the semaphore has been placed into *HEV* already. *ulTimeout* is the number of milliseconds that the calling thread will wait for the event semaphore to be posted. If the specified event semaphore is not posted during this time interval, the request times out.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSERRORS       /* error codes      */

#include <os2.h>
#include <stdio.h>

HEV     hev;        /* Event semaphore handle         */
ULONG   ulTimeout;  /* Number of milliseconds to wait */
APIRET  ulrc;       /* Return code                    */

ulTimeout = 60000;  /* Wait for a maximum of 1 minute */

ulrc = DosWaitEventSem(hev,
                       ulTimeout);

if (ulrc == ERROR_TIMEOUT) {
    printf("DosWaitEventSem call timed out");
    return;
}

if (ulrc == ERROR_INTERRUPT) {
    printf("DosWaitEventSem call was interrupted");
    return;
}
```

```
if (ulrc != 0) {
    printf("DosWaitEventSem error: return code = %ld",
           ulrc);
    return;
}
```

If the time limit specified in *ulTimeout* is reached before the semaphore is posted, ERROR_TIMEOUT is returned. If the waiting period is interrupted for some reason before the semaphore is posted, ERROR_INTERRUPT is returned. If SEM_IMMEDIATE_RETURN is specified for the time limit, DosWaitEventSem returns to the calling thread immediately. If SEM_INDEFINITE_WAIT is specified for the time limit, the thread waits indefinitely.

Unlike multiple event semaphores in a muxwait list, which are level-triggered, single event semaphores are *edge-triggered*. This means that if an event semaphore is posted and then reset before a waiting thread gets a chance to run, the semaphore is considered to be posted for the rest of that thread's waiting period; the thread does not have to wait for the semaphore to be posted again.

Querying an Event Semaphore

DosQueryEventSem returns the current post count of a semaphore. The post count is the number of times that the semaphore has been posted (with DosPostEventSem) since the last time the semaphore was reset. A count of 0 indicates that the semaphore is in the reset state; therefore, OS/2 will block any threads that call DosWaitEventSem to wait on the semaphore.

Any thread in the process that created an event semaphore can obtain the post count for the semaphore by calling DosQueryEventSem. Threads in other processes can also call DosQueryEventSem, but they must first gain access to the semaphore by calling DosOpenEventSem.

The following code fragment retrieves the post count for an event semaphore. Assume that the handle of the semaphore has been placed into *HEV* already.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HEV     hev;       /* Event semaphore handle                          */
ULONG   ulPostCt;  /* Current post count for the semaphore (returned) */
APIRET  ulrc;      /* Return code                                     */

ulrc = DosQueryEventSem(hev,
                        &ulPostCt);

if (ulrc != 0) {
    printf("DosQueryEventSem error: return code = %ld",
           ulrc);
    return;
}
```

If the specified event semaphore does not exist, ERROR_INVALID_HANDLE is returned.

Using Mutex Semaphores

An application can use a mutual exclusion (mutex) semaphore to protect a shared resource from simultaneous access by multiple threads or processes. For example, if several processes must write to the same disk file, the mutex semaphore ensures that only one process at a time writes to the file.

Creating a Mutex Semaphore

Mutex semaphores are created by calling DosCreateMutexSem. This function also opens the semaphore for the calling process and its

threads.

When a mutex semaphore is created, a flag is set to specify the initial state of the semaphore, owned or unowned. If the semaphore is owned by a thread, other threads requesting the semaphore are blocked. If the semaphore is unowned-not owned by any thread- then any thread requesting ownership will be granted ownership immediately.

If the calling thread sets the initial state to owned, it owns the semaphore as soon as OS/2 creates the semaphore and can proceed to access the resource that the semaphore was created to protect.

If the semaphore is unowned, any thread in the creating process can subsequently request ownership of the semaphore by calling DosRequestMutexSem. Threads in other processes can gain ownership of the semaphore, but they must call DosOpenMutexSem to acquire access to the semaphore before they can call DosRequestMutexSem.

Mutex semaphores can be defined as either private or shared.

- Private semaphores are always unnamed and are therefore identified by their handles. They can be used only by threads within a single process.
- Shared semaphores can be either named or unnamed. If named, they can be opened using either the name or the handle. The handle returned by DosOpenMutexSem is then used to identify the semaphore for all other functions. Semaphore names must include the prefix \SEM32\ and must conform to file system naming conventions. Shared semaphores can be used by threads in multiple processes.

The following code fragment creates a mutex semaphore:

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>

HMTX hmtxProtFile;

DosCreateMutexSem("\\sem32\\ProtFile",          /* Named-shared semaphore */
                  &hmtxProtFile,
                  0,
                  FALSE);                       /* Initially unowned      */
    .
    .    /* Get data to write to shared file. */
    .
```

There is a system-wide limit of 65536 shared semaphores (including mutex, event, and muxwait semaphores); in addition, each process can have up to 65536 private semaphores.

OS/2 maintains a usage count for each semaphore. DosCreateMutexSem initializes the usage count to 1. Thereafter, each call to DosOpenMutexSem increments the count, and each call to DosCloseMutexSem decrements it.

Opening a Mutex Semaphore

All of the threads belonging to the process that creates a mutex semaphore have immediate access to the semaphore. Threads in other processes must request access to the semaphore by calling DosOpenMutexSem before they can use the semaphore in other mutex semaphore functions.

Access to system resources is granted on a per-process basis. Therefore, a semaphore that has been opened by one thread in a process is open to all other threads in that process as well.

DosOpenMutexSem merely provides access to a mutex semaphore. To request ownership of a mutex semaphore, a thread must call DosRequestMutexSem.

When a process no longer requires access to a mutex semaphore, it should close the semaphore by calling DosCloseMutexSem. However, if a process ends without closing an open semaphore, the semaphore is closed by OS/2.

Each call to DosOpenMutexSem. increments the usage count of the semaphore. This count is initialized to 1 when the semaphore is created and is decremented by each call to DosCloseMutexSem. When the usage count reaches 0, the semaphore is deleted by the system.

Calls to DosOpenMutexSem and DosCloseMutexSem. can be nested, but the usage count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_OPENS is returned.

If a process ends without releasing a mutex semaphore that it owns, any other thread that subsequently tries to open the semaphore will receive ERROR_SEM_OWNER_DIED. This return code indicates that the owning process ended abnormally, leaving the protected resource in an indeterminate state. However, the semaphore is still opened for the calling thread, enabling the thread to call DosQueryMutexSem to find out which process ended without releasing the semaphore. The thread can then take appropriate action concerning the semaphore and

the protected resource.

Requesting a Mutex Semaphore

In order to access a shared resource, a process must own the mutex semaphore that is protecting the shared resource. Ownership is obtained by first opening the mutex semaphore with DosOpenMutexSem, then using DosRequestMutexSem to request ownership of the semaphore. If another process already owns the semaphore, the requesting process is blocked. If the semaphore is not owned, OS/2 grants ownership to the requesting process and the process can access the shared resource. When the process is finished using the shared resource, it uses DosReleaseMutexSem to relinquish its ownership of the semaphore, thereby enabling another process to gain ownership.

A process can gain ownership of a mutex semaphore in three ways:

1. The thread that creates a mutex semaphore can designate itself as the owner by setting a flag when it calls DosCreateMutexSem.
2. Any thread in the process that created the semaphore can request ownership by calling DosRequestMutexSem.
3. A thread in another process must request access to the semaphore with DosOpenMutexSem before it can call DosRequestMutexSem.

Note that ownership of a mutex semaphore is given only to the requesting thread; it is not shared by other threads in the same process.

If a mutex semaphore is unowned, DosRequestMutexSem sets it as *owned* and returns immediately to the caller. If the semaphore is already owned, the calling thread is blocked until either the owning thread calls DosReleaseMutexSem to release the semaphore, or a specified time limit is reached.

The following code fragment shows how a process opens a mutex semaphore, requests it, and, after writing to the shared file, releases and closes the semaphore:

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>

HMTX hmtxProtFile;

DosOpenMutexSem("\\sem32\\ProtFile",
                &hmtxProtFile);          /* Opens for this process    */

DosRequestMutexSem(hmtxProtFile,
                   5000);                /* Returns in 5 seconds if   */
    .                                    /* Ownership not obtained    */
    .
    .                                    /* Write data to shared file */
    .
    .
DosReleaseMutexSem(hmtxProtFile);        /* Releases ownership        */
    .
    .                                    /* Continue execution        */
    .
    .
DosCloseMutexSem(hmtxProtFile);          /* Finished with shared file */
```

If more than one thread is blocked on a DosRequestMutexSem request, the thread with the highest priority level is the first to be unblocked and given ownership of the semaphore. If more than 1 of the waiting threads have the same priority level, then FIFO ordering is used to determine which thread is unblocked and given ownership.

The *time-out* parameter (5000 milliseconds in the example above) places a limit on the amount of time a thread blocks on a DosRequestMutexSem request. If the time limit is reached before the thread gains ownership of the semaphore, ERROR_TIMEOUT is returned. If SEM_IMMEDIATE_RETURN is specified for the time limit, DosRequestMutexSem returns without blocking the thread. The thread can then perform other operations and call DosRequestMutexSem again later if it still requires access to the protected resource. If SEM_INDEFINITE_WAIT is specified for the time limit, the thread waits indefinitely. If the thread is unblocked by an external event while it is waiting for the mutex semaphore (as when a *No Wait* I/O request has just been completed), ERROR_INTERRUPT is returned to the caller.

In addition to the usage count that OS/2 maintains for all semaphores, OS/2 maintains a request count for each mutex semaphore. Each call to DosRequestMutexSem increments the count, and each call to DosReleaseMutexSem decrements it.

Calls to DosRequestMutexSem and DosReleaseMutexSem can be nested, but the request count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_SEM_REQUESTS is returned. When calls to DosRequestMutexSem and DosReleaseMutexSem are nested, a call to DosReleaseMutexSem merely decrements the request count for the semaphore; the semaphore is not actually released to another thread until its request count is 0. If a process ends while it owns a mutex semaphore, all of the currently blocked DosRequestMutexSem requests, as well as any future requests for the semaphore, return ERROR_SEM_OWNER_DIED. This return code indicates that the owning process ended abnormally, leaving the protected resource in an indeterminate state. An application that

receives this error should close the mutex semaphore (so that it can be deleted from OS/2), because it is no longer valid. Appropriate action should also be taken concerning the protected resource.

Releasing a Mutex Semaphore

A thread can release ownership of a mutex semaphore by calling DosReleaseMutexSem. Each call to DosReleaseMutexSem decrements the request count that is maintained for the semaphore by OS/2. Each call to DosRequestMutexSem increments the count.

The following code fragment relinquishes ownership of a mutex semaphore. Assume that the handle of the semaphore has been placed into *hmtx* already.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HMTX    hmtx;    /* Mutex semaphore handle */
APIRET  ulrc;    /* Return code            */

ulrc = DosReleaseMutexSem(hmtx);

if (ulrc != 0) {
    printf("DosReleaseMutexSem error: return code = %ld",
           ulrc);
    return;
}
```

Calls to DosRequestMutexSem and DosReleaseMutexSem can be nested, but the request count cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_SEM_REQUESTS is returned. When calls to DosRequestMutexSem and DosReleaseMutexSem are nested, a call to DosReleaseMutexSem merely decrements the request count for the semaphore; the semaphore is not actually released to another thread until its request count is 0.

Closing a Mutex Semaphore

When a process no longer requires access to a mutex semaphore, it can close the semaphore by calling DosCloseMutexSem. However, if a process ends without closing an open semaphore, the semaphore is closed by OS/2.

The following code fragment closes a mutex semaphore. Assume that the handle of the semaphore has been placed into *hmtx* already.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HMTX    hmtx;    /* Mutex semaphore handle */
APIRET  ulrc;    /* Return code            */

ulrc = DosCloseMutexSem(hmtx);

if (ulrc != 0) {
    printf("DosCloseMutexSem error: return code = %ld",
           ulrc);
    return;
}
```

Each call to DosCloseMutexSem decrements the usage count of the semaphore. This count is initialized to 1 when the semaphore is created and is incremented by each call to DosOpenMutexSem. When the usage count reaches 0, the semaphore is deleted by OS/2.

The call to DosCloseMutexSem that decrements the usage count to 0 and causes the semaphore to be deleted is referred to as the *final* close. The final close will not succeed if either of the following conditions exists:

- The semaphore is owned by another thread in the same process.
- Another thread in the same process is still blocked on a DosRequestMutexSem request for the semaphore.

For both conditions, ERROR_SEM_BUSY is returned.

ERROR_SEM_BUSY is also returned if a thread tries to close a mutex semaphore that it still owns. The thread must first relinquish ownership of the semaphore by calling DosReleaseMutexSem.

Calls to DosOpenMutexSem and DosCloseMutexSem can be nested, but the usage count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_OPENS is returned.

Querying a Mutex Semaphore

An application can use DosQueryMutexSem to determine the current owner of a mutex semaphore and to obtain a count of the number of requests on it. If the mutex semaphore is not owned, the request count is 0.

Any thread in the process that created a mutex semaphore can obtain information about the semaphore by calling DosQueryMutexSem. Threads in other processes can also call DosQueryMutexSem, but they must first gain access to the semaphore by calling DosOpenMutexSem.

If the mutex semaphore exists and is owned, DosQueryMutexSem returns the process and thread identifications of the owner, as well as the request count for the semaphore. The request count is the number of DosRequestMutexSem calls minus the number of DosReleaseMutexSem calls that have been made for the semaphore by the owning thread.

If DosQueryMutexSem returns a request count of 0, the mutex semaphore is unowned.

If the owning process ended without calling DosCloseMutexSem, then ERROR_SEM_OWNER_DIED is returned, and the output parameters contain information about the ended owning process.

Using Muxwait Semaphores

A process that requires exclusive use of several shared resources at once can use a multiple wait (muxwait) semaphore to obtain ownership of all the mutex semaphores protecting the shared resources. A process can also use a muxwait semaphore to wait on a group of event semaphores so that the process continues running whenever events of interest occur.

A muxwait semaphore can refer to up to 64 event or mutex semaphores. An application cannot refer to event and mutex semaphores in a single muxwait semaphore, or include a muxwait semaphore in another muxwait semaphore.

Creating a Muxwait Semaphore

DosCreateMuxWaitSem is used to create muxwait semaphores. This function also opens (obtains access to) the semaphore for the calling process and its threads. Threads in other processes must call DosOpenMuxWaitSem to open the semaphore before they can use it in any other muxwait semaphore function.

All the semaphores in the muxwait list must be created and opened before the muxwait list can be created.

The following code fragment creates five event semaphores and a corresponding array of semaphore records. The array is used to specify the semaphores included in the muxwait semaphore in the subsequent call to DosCreateMuxWaitSem.

```
#define INCL_DOSSEMAPHORES   /* DOS semaphore values */
#define INCL_DOSERRORS       /* DOS error values */
#include <os2.h>
#include <stdio.h>

int main(VOID) {
 HMUX      hmuxHandAny = NULLHANDLE;      /* Muxwait handle    */
 HEV       hev[5]      = {0};             /* Event semaphores  */
 SEMRECORD apsr[5]     = {{0}};           /* Semaphore records */
 APIRET    ulrc        = NO_ERROR;        /* Return code       */
 ULONG     ulLoop      = 0;               /* Loop count        */
 ULONG     ulSem       = 0;
```

```
for (ulLoop = 0; ulLoop < 5; ulLoop++) {
    ulrc = DosCreateEventSem((PSZ) NULL,
                             &hev[ulLoop],
                             0,
                             FALSE);

    if (ulrc != NO_ERROR) {
      printf("DosCreateEventSem error:  return code = %u\n",
             ulrc);
      return 1;
    }
    apsr[ulLoop].hsemCur = (HSEM) hev[ulLoop],
    apsr[ulLoop].ulUser = 0;
} /* end for */

ulrc = DosCreateMuxWaitSem((PSZ) NULL,
                           &hmuxHandAny,
                           5L,             /* Number of semaphores in list */
                           apsr,           /* Semaphore list               */
                           DCMW_WAIT_ANY);  /* Wait for any semaphore       */

    if (ulrc != NO_ERROR) {
      printf("DosCreateMuxWaitSem error:  return code = %u\n",
             ulrc);
      return 1;
    }


ulrc = DosWaitMuxWaitSem(hmuxHandAny,
                         SEM_IMMEDIATE_RETURN,
                         &ulSem);          /* No semaphores have been posted, so
                                              we should see a timeout below...    */

    if (ulrc != ERROR_TIMEOUT) {
      printf("DosWaitMuxWaitSem error:  return code = %u\n",
             ulrc);
      return 1;
    }

ulrc = DosDeleteMuxWaitSem(hmuxHandAny,
                           apsr[0].hsemCur);

    if (ulrc != NO_ERROR) {
      printf("DosDeleteMuxWaitSem error:  return code = %u\n",
             ulrc);
      return 1;
    }

ulrc = DosCloseMuxWaitSem(hmuxHandAny);
     if (ulrc != NO_ERROR) {
       printf("DosCloseMuxWaitSem error: return code = %u\n",
              ulrc);
       return 1;
     }

return NO_ERROR;
}
```

Muxwait semaphores can be defined as either private or shared:

- Private semaphores are always unnamed and are therefore always identified by their handles. They can be used only by threads within a single process.
- Shared semaphores can be either named or unnamed. If named, they can be opened using either the name or the handle. The handle returned by DosOpenMuxWaitSem is then used to identify the semaphore for all other functions. Semaphore names must include the prefix \SEM32\ and must conform to file system naming conventions. Shared semaphores can be used by threads in multiple processes.

There is a system-wide limit of 65536 (64K) shared semaphores (including mutex, event, and muxwait semaphores); in addition, each process can have up to 65536 (64K) private semaphores.

The following conditions apply to the kinds of semaphores that can be included in a muxwait-semaphore list:

- The list must contain either mutex semaphores or event semaphores. It cannot contain both at the same time and it cannot contain other muxwait semaphores.
- If the muxwait semaphore is shared, then all the semaphores in the list must also be shared.
- If the muxwait semaphore is private, then the semaphores in its list can be either private or shared.

If any of these conditions is violated, ERROR_WRONG_TYPE is returned.

The muxwait list can contain a maximum of 64 event semaphores or mutex semaphores. If an attempt is made to exceed this maximum, ERROR_TOO_MANY_SEMAPHORES is returned.

If the owners of any of the *mutex* semaphores in the muxwait semaphore list have ended without releasing them, ERROR_SEM_OWNER_DIED is returned. The thread should call DosQueryMutexSem for each mutex semaphore in the muxwait-semaphore list so that it can determine which semaphores are in the *Owner Died* state.

Each mutex semaphore that returns ERROR_SEM_OWNER_DIED from the query should be closed by calling DosCloseMutexSem. Also, because semaphore handles can be reused, the mutex semaphores that are closed must be deleted from the muxwait-semaphore list by calling DosDeleteMuxWaitSem.

OS/2 maintains a usage count for each semaphore. DosCreateMuxWaitSem initializes the usage count to 1. Thereafter, each call to DosOpenMuxWaitSem increments the count, and each call to DosCloseMuxWaitSem decrements it.

One parameter of this function is a pointer to an array of SEMRECORD data structures. Each data structure contains one semaphore record for each of the semaphores to be included in the muxwait semaphore. A semaphore record contains the handle and a programmer-defined identifier for that semaphore.

Opening a Muxwait Semaphore

Processes other than the semaphore-creating process must use DosOpenMuxWaitSem to gain access to the muxwait semaphore before they can use the semaphore in any other muxwait semaphore function. All of the threads that belong to the process that creates the muxwait semaphore have immediate access to the semaphore.

The following code fragment opens a system muxwait semaphore.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>
#include <string.h>

UCHAR   ucName[40];  /* Semaphore name           */
HMUX    hmux;        /* Muxwait semaphore handle */
APIRET  ulrc;        /* Return code              */

strcpy(ucName,
       "\\SEM32\\MUXWAIT1");  /* Name of the system muxwait semaphore */

ulrc = DosOpenMuxWaitSem(ucName,
                         &hmux);

if (ulrc != 0) {
    printf("DosOpenMuxWaitSem error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, *hmux* contains the handle of the system muxwait semaphore.

Opening a muxwait semaphore does not open the semaphores in its muxwait list. A process must open each of the semaphores included in a muxwait semaphore before it opens the muxwait semaphore. Otherwise, DosOpenMuxWaitSem returns the ERROR_INVALID_HANDLE error value to the calling function.

Access to semaphores is on a per-process basis. Therefore, a semaphore that has been opened by one thread in a process is open to all other threads in that process as well.

Note that DosOpenMuxWaitSem merely provides access to a muxwait semaphore. In order to wait for a muxwait semaphore to clear, a thread must call DosWaitMuxWaitSem.

When a process no longer requires access to a muxwait semaphore, it closes the semaphore by calling DosCloseMuxWaitSem. However, if a process ends without closing an open semaphore, the semaphore is closed by OS/2.

Each call to DosOpenMuxWaitSem increments the usage count of the semaphore. This count is initialized to 1 when the semaphore is created and is decremented by each call to DosCloseMuxWaitSem. When the usage count reaches 0, the semaphore is deleted by OS/2.

Calls to DosOpenMuxWaitSem and DosCloseMuxWaitSem can be nested, but the usage count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_OPENS is returned.

Even if the owner of a mutex semaphore in a muxwait-semaphore list has ended without releasing the semaphore, the muxwait semaphore is still opened. Subsequent calls to the muxwait semaphore will return ERROR_SEM_OWNER_DIED. But because the process has opened the semaphore, it can then call DosQueryMuxWaitSem to identify all the mutex semaphores in the muxwait list. Next, the process can call DosQueryMutexSem for each mutex semaphore in the list to find out which ones are in the *Owner Died* state. Each mutex semaphore that returns ERROR_SEM_OWNER_DIED from the query should be closed by calling DosCloseMutexSem. Also, because semaphore handles can be reused, the mutex semaphores that are closed should be deleted from the muxwait-semaphore list by calling DosDeleteMuxWaitSem.

Closing a Muxwait Semaphore

When a process no longer requires access to a muxwait semaphore, it closes the semaphore by calling DosCloseMuxWaitSem. However, if a process ends without closing an open semaphore, the semaphore is closed by OS/2.

Each call to DosCloseMuxWaitSem decrements the usage count of the semaphore. This count is initialized to 1 when the semaphore is created and is incremented by each call to DosOpenMuxWaitSem. When the usage count reaches 0, the semaphore is deleted by OS/2.

The call to DosCloseMuxWaitSem that decrements the usage count to 0 and causes the semaphore to be deleted is referred to as the *final close* . If a thread attempts to perform the final close for a semaphore while another thread in the same process is still waiting for it, ERROR_SEM_BUSY is returned.

Calls to DosOpenMuxWaitSem and DosCloseMuxWaitSem can be nested, but the usage count for a semaphore cannot exceed 65535. If an attempt is made to exceed this number, ERROR_TOO_MANY_OPENS is returned.

Waiting for a Muxwait Semaphore

A thread can wait on a muxwait semaphore by using DosWaitMuxWaitSem.

Any thread in the process that created a muxwait semaphore can wait for the semaphore to clear by calling DosWaitMuxWaitSem. Threads in other processes can also call DosWaitMuxWaitSem, but they must first gain access to the semaphore by calling DosOpenMuxWaitSem.

The following code fragment waits for a muxwait semaphore to clear. Assume that the handle of the semaphore has been placed into *hmux* already. *ulTimeout* is the number of milliseconds that the calling thread will wait for the muxwait semaphore to clear. If the specified muxwait semaphore is not cleared during this time interval, the request times out.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#include <os2.h>
#include <stdio.h>

HMUX     hmux;      /* Muxwait semaphore handle              */
ULONG    ulTimeout; /* Number of milliseconds to wait        */
ULONG    ulUser;    /* User field for the semaphore that was */
                    /* posted or released (returned)         */
APIRET   ulrc;      /* Return code                           */

ulTimeout = 60000;  /* Wait for a maximum of 1 minute        */

ulrc = DosWaitMuxWaitSem(hmux,
                         ulTimeout,
                         &ulUser);

if (ulrc == ERROR_TIMEOUT) {
    printf("DosWaitMuxWaitSem call timed out");
    return;
}

if (ulrc == ERROR_INTERRUPT) {
    printf("DosWaitMuxWaitSem call was interrupted");
    return;
}

if (ulrc != 0) {
    printf("DosWaitMuxWaitSem error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, the *ulUser* variable contains the user identifier of the semaphore that caused the wait to terminate. If the caller had to wait for all the semaphores within the muxwait semaphore to clear, then the value corresponds to the last semaphore within the muxwait semaphore to clear. If the caller had to wait for any semaphore with the muxwait semaphore to clear, then the value corresponds to that semaphore.

An application can use the DCMW_WAIT_ANY flag in DosCreateMuxWaitSem to block a thread until any one of the event or mutex semaphores included in the muxwait semaphore is posted or released. If the muxwait semaphore refers to mutex semaphores, the thread only gains ownership of the one mutex semaphore that was released.

An application can use the DCMW_WAIT_ALL flag in DosCreateMuxWaitSem to block a thread until all of the event or mutex semaphores included in the muxwait semaphore are posted or released. If the muxwait semaphore refers to mutex semaphores, the thread does not gain ownership of any of the mutex semaphores until they are all released. When all are released, the thread becomes owner of all the mutex semaphores included in the muxwait semaphore. If the muxwait semaphore refers to event semaphores, the thread will not run until all of the event semaphores are in the posted state at the same time. This is because event semaphores in a muxwait list are *level-triggered*, unlike individual event semaphores, which are edge-triggered.

For example, suppose that a thread is waiting for five event semaphores in a muxwait list to be posted. The first semaphore is posted and then reset. Next, the remaining semaphores are all posted, and they remain in the posted state. The thread that is waiting for the muxwait semaphore will not run until the first semaphore is posted again.

If an application specifies the DCMW_WAIT_ANY flag when the semaphore is created, DosWaitMuxWaitSem returns the programmer-defined identifier of the semaphore that is subsequently posted or released. If an application specifies the DCMW_WAIT_ALL flag, DosWaitMuxWaitSem returns the programmer-defined identifier of the last semaphore that was posted or released.

The *ulTimeout* parameter places a limit on the amount of time a thread blocks on a DosWaitMuxWaitSem request. If the time limit is reached before the semaphore has cleared, ERROR_TIMEOUT is returned. If SEM_IMMEDIATE_RETURN is specified as the time limit, DosWaitMuxWaitSem returns without blocking the thread. The thread can then go on to perform other operations and call DosWaitMuxWaitSem again later to wait for the event or mutex semaphores in the muxwait list to be posted or released. If a time limit of SEM_INDEFINITE_WAIT is specified, the thread waits (is blocked) indefinitely. If the thread is unblocked by an external event while it is waiting for the muxwait semaphore (as when a "no wait" I/O request has just been completed), DosWaitMuxWaitSem returns ERROR_INTERRUPT.

When a thread is waiting for any one of the semaphores in a muxwait list to be posted or released, the semaphores are checked in the order in which they are defined in the list.

Waiting for Multiple Event Semaphores
The following information pertains only to muxwait semaphores that consist of multiple *event* semaphores:

Unlike individual event semaphores, which are edge-triggered, event semaphores in a muxwait list are *level-triggered*. This means that if a thread is waiting for all of the event semaphores in the muxwait list, it will not run until all of the event semaphores are in the posted state at the same time.

For example, a thread is waiting for five event semaphores in a muxwait list to be posted. The first semaphore is posted and then reset. Next, the remaining semaphores are all posted, and they remain in the posted state. The thread that is waiting for the muxwait semaphore will not run until the first semaphore is posted again.

Waiting for Multiple Mutex Semaphores
The following information pertains only to muxwait semaphores that consist of multiple *mutex* semaphores:

- If a thread is waiting for all of the mutex semaphores in a muxwait list to be released, it does not receive ownership of any of the semaphores until all of the semaphores have been released.

- If a thread is waiting for any one of the mutex semaphores in a muxwait list, then the thread gains ownership only of the first mutex semaphore that is released. The ownership of all other mutex semaphores in the muxwait list remains unchanged.

- If two threads have the same priority, then a thread that is waiting for a mutex semaphore in a muxwait list takes precedence over a thread that has requested ownership of only the individual semaphore, provided all other mutex semaphores in the muxwait list have been released. For example, a mutex semaphore that is part of a muxwait semaphore is released. One thread has requested ownership of that single mutex semaphore, and another thread with the same priority is waiting for the muxwait semaphore that contains the same mutex semaphore. If all of the other mutex semaphores in the muxwait list are *unowned* and ready to be given to the muxwait semaphore, then the thread that is waiting for the muxwait semaphore will run first.

- If the owners of any of the mutex semaphores in the muxwait semaphore list have ended without releasing them, ERROR_SEM_OWNER_DIED is returned. The thread must then call DosQueryMuxWaitSem to obtain the records of all the semaphores in the muxwait list. Next, the thread must call DosQueryMutexSem for each mutex semaphore in the muxwait-semaphore list so that it can determine which semaphores are in the *Owner Died* state.

 Each mutex semaphore that returns ERROR_SEM_OWNER_DIED from the query should be closed by calling DosCloseMutexSem. Also, because semaphore handles can be reused, the mutex semaphores that are closed should be deleted from the muxwait-semaphore list by calling DosDeleteMuxWaitSem.

- If any of the mutex semaphores in the muxwait list are owned by the calling thread, ERROR_MUTEX_OWNED is returned.

Adding a Semaphore to a Muxwait List

An application uses DosAddMuxWaitSem to add semaphores to a muxwait semaphore that has already been created, even while threads are waiting on the muxwait semaphore.

Any thread in the process that created a muxwait semaphore can add a mutex semaphore or an event semaphore to the muxwait list by calling DosAddMuxWaitSem. Threads in other processes can also use this function, but they must first gain access to the semaphore by calling DosOpenMuxWaitSem.

A maximum of 64 semaphores can be included in a muxwait-semaphore list. If an attempt is made to exceed this maximum, ERROR_TOO_MANY_SEMAPHORES is returned.

All of the semaphores in a muxwait-semaphore list must be of the same type. That is, if a mutex semaphore is being added, then the other semaphores in the list must be mutex semaphores. If an event semaphore is being added, then the other semaphores in the list must be event semaphores. A shared muxwait semaphore can contain only shared semaphores in its list. A private muxwait semaphore can contain both private and shared semaphores. If any of these conditions is violated, ERROR_WRONG_TYPE is returned.

If the semaphore is successfully added to the muxwait list, DosAddMuxWaitSem checks to see whether each thread that is waiting for the muxwait semaphore has the newly added semaphore open in its process. The muxwait semaphore is invalid for any waiting threads that do not have the newly added semaphore open in their process; these threads are unblocked with a return code of ERROR_INVALID_HANDLE. Any processes that opened the muxwait semaphore before the add operation and that do not have the new semaphore open, will have to open the new semaphore before making any further use of the muxwait semaphore. Any future calls concerning the muxwait semaphore by processes that do not have the new semaphore open will have ERROR_INVALID_HANDLE returned until the new semaphore is opened.

A thread that receives a return code of ERROR_INVALID_HANDLE can take the following corrective action:

1. First, the thread can obtain the records of all the semaphores in the muxwait list by calling DosQueryMuxWaitSem.
2. Next, it can query each semaphore in the muxwait list, using DosQueryMutexSem or DosQueryEventSem, to find out which semaphore is not open to its process.
3. Finally, it can open the semaphores that are not open by calling DosOpenMutexSem or DosOpenEventSem.

As soon as this semaphore is opened, the muxwait semaphore becomes valid again for the process, as long as no other changes have been made to the muxwait list to make it invalid. However, in order to successfully wait for the muxwait semaphore, the process must call DosWaitMuxWaitSem again.

A semaphore must be open for a process before the process can add that semaphore to a muxwait semaphore. If it is not open and a thread is waiting on the muxwait semaphore, DosAddMuxWaitSem returns ERROR_INVALID_HANDLE to the process adding the new semaphore, and the waiting thread continues waiting.

Deleting a Semaphore from a Muxwait List

An application can delete semaphores from a muxwait semaphore by using DosDeleteMuxWaitSem.

Any thread in the process that created a muxwait semaphore can delete a mutex or event semaphore from the muxwait list by calling DosDeleteMuxWaitSem. Threads in other processes can also use this function, but they must first gain access to the semaphore by calling DosOpenMuxWaitSem.

Semaphores can be deleted from the muxwait list even while threads are currently waiting for the semaphore. If the deleted semaphore is the only one in the muxwait list that has not yet been posted or released, then threads that are waiting for the muxwait semaphore are unblocked. Also, if the deleted semaphore happens to be the last one that a particular thread was waiting for, that thread is unblocked. Also, if the deleted semaphore is the last one in the muxwait list (that is, if the list is now empty), then all the threads that are waiting for the muxwait semaphore are unblocked.

Querying a Muxwait Semaphore

Processes use DosQueryMuxWaitSem to obtain the semaph ore records for each of the semaphores included in the muxwait semaphore.

Any thread in the process that created a muxwait semaphore can obtain information about the semaphores in the muxwait list by calling DosQueryMuxWaitSem. Threads in other processes can also use this function, but they must first gain access to the semaphore by calling DosOpenMuxWaitSem.

An application must provide this function with an array in which to store the semaphore records. If the array is not large enough to hold all of the semaphore records that are in the muxwait list, then ERROR_PARAM_TOO_SMALL is returned, and the record-counting parameter of DosQueryMuxWaitSem will contain the number of semaphore records that are in the muxwait list. The calling thread can then allocate the correct amount of space and call DosQueryMuxWaitSem again with the correct amount of space for the list of records.

If the owner of any mutex semaphore in the muxwait-semaphore list has ended without releasing the semaphore, the records of all the semaphores in the list are still returned, but DosQueryMuxWaitSem also returns ERROR_SEM_OWNER_DIED. The calling thread can call DosQueryMutexSem for each mutex semaphore in the muxwait-semaphore list so that it can determine which semaphores are in the *Owner Died* state. The process can then close the unowned mutex semaphores.

Each mutex semaphore that returns ERROR_SEM_OWNER_DIED from the query should be closed by calling DosCloseMutexSem. Also, because semaphore handles can be reused, the mutex semaphores that are closed should be deleted from the muxwait-semaphore list by calling DosDeleteMuxWaitSem.

If the specified muxwait semaphore does not exist, ERROR_INVALID_HANDLE is returned.

Timers

This chapter describes how to create and use timers. Timers enable applications to time events by waiting for an interval to elapse or by waiting for a semaphore to be posted.

The following topics are related to the information in this chapter:

- Program execution and control
- Semaphores

About Timers

Because OS/2 is a multitasking system, an application cannot predict when it will lose execution control or how much time will elapse before control returns. A *timer* enables an application to suspend operation for a specific length of time, to block a thread until an interval has elapsed, or to post an event semaphore at repeated intervals.

Timers are managed by OS/2. When an application requests a timer, the system monitors the system clock and notifies the application when the interval has elapsed.

The system clock counts the number of system-clock interrupts (clock ticks) that have occurred since the system was started. On most hardware, clock ticks occur approximately 32 times a second, so the length of a tick is approximately 31.25 milliseconds.

When an application specifies a timer interval, the system rounds up the interval to the next clock tick. For example, if an application requests a 10 millisecond interval, it will sleep for at least 31.25 milliseconds. If an application requests a 100 millisecond interval, the actual interval will be at least 125 milliseconds (4 ticks).

Because OS/2 is a preemptive operating system, there is no guarantee that a thread will resume immediately after the timer interval. If a higher priority process or thread is executing, the timed thread must wait.

Although timers are not absolutely accurate, they can be used where the inaccuracy can be ignored. In a real-time control application, for example, an event can be timed in seconds or minutes, so an error of a few milliseconds is unimportant. If an application requires as much accuracy as the system can provide, it can dedicate a thread to managing timer intervals and then elevate the priority of that thread.

Suspending Threads

An application can use DosSleep to suspend operation of a thread for a specified interval. The system waits the specified number of milliseconds (subject to the round-off error just discussed) before returning control to the application. Because a sleeping application yields execution control to the system, the system can execute other processes or threads while the application sleeps.

The following code fragment shows how to suspend the calling thread for one minute:

```
#define INCL_DOSPROCESS   /* Process and thread values */
```

```
#include <os2.h>
#include <stdio.h>

ULONG   ulTimeInterval;   /* Interval in milliseconds  */
APIRET  ulrc;             /* Return code               */

ulTimeInterval = 60000;

ulrc = DosSleep(ulTimeInterval);

if (ulrc != 0) {
    printf("DosSleep error: return code = %ld",
           ulrc);
    return;
}
```

See Suspending the Current Thread for more information on DosSleep.

Asynchronous Timers

DosSleep is useful for temporarily suspending a thread but is much less useful for timing. Typically, an application carries out part of its task and then waits an interval. If the execution time varies (as it will if the application runs on different hardware), the overall interval varies. In these situations, asynchronous timers provide greater precision than DosSleep.

OS/2 supports two types of asynchronous timers, single-interval (one-shot) and repeated. DosAsyncTimer starts a single-interval timer. During the timing interval, the application can carry out other tasks. The system posts an event semaphore when the timing interval elapses. The application can reset the semaphore with DosResetEventSem before starting the timer. DosAsyncTimer yields a more accurate timing interval than DosSleep because the interval is independent of the execution time.

DosStartTimer starts a repeated timer. The system posts an event semaphore each time the interval expires. The application can reset the semaphore before starting the timer and after each posting. When the application resets the semaphore with DosResetEventSem, it can check the *cPosts* value to determine how many times the semaphore has been posted. If the semaphore has been posted more than once, the application has missed a timer interval.

Using Timers

Applications frequently need to synchronize the execution of threads, to cause an event to occur after a specified interval, or to cause an event to occur at regular intervals. Timers are typically used to enable an application to pause before processing user input or to carry out a task at a given time.

OS/2 provides the following timer functions:

- DosSleep suspends the execution of the calling thread, enabling other threads to run while the calling thread sleeps.
- DosAsyncTimer starts a single-interval timer.
- DosStartTimer starts a repeated-interval timer.
- DosStopTimer stops a single-interval or repeated-interval timer.

The system also provides two functions, DosGetDateTime and DosSetDateTime, for getting and setting the system date and time.

The timers that are started by DosAsyncTimer and DosStartTimer are asynchronous timers; that is, the timers run independently of the calling thread, enabling the calling thread to perform other operations while the timer is running. When an asynchronous timer interval expires, the system notifies the application by posting an event semaphore.

Time intervals for DosAsyncTimer, DosStartTimer, and DosSleep are specified in milliseconds; however, it is important to recognize that the actual duration of the specified time interval will be affected by two factors:

- First, the system clock keeps track of time in less precise units known as *clock ticks* . On most hardware, clock ticks occur approximately 32 times a second, so each tick interval lasts approximately 31.25 milliseconds. (To determine the duration of a clock tick on your computer, call DosQuerySysInfo, and examine the timer-interval field.)

 Because clock ticks are less precise than millisecond values, any time interval that is specified in milliseconds will essentially be

rounded up to the next clock tick.

- Second, because OS/2 is a priority-based, multitasking operating system, there is no guarantee that a thread will resume execution immediately after the timer interval expires. If a higher priority process or thread is running, or if a hardware interrupt occurs, the timed thread blocks. (To minimize delays caused by preemptive scheduling, an application can dedicate a thread to managing time-critical tasks, and then raise that thread to a higher priority.)

These factors usually cause the timer interval to be longer than requested; however, it will generally be within a few clock ticks.

Timers for Presentation Manager applications are provided through the message queue. Therefore, a Presentation Manager application will not use the timer functions unless it performs some real-time control task.

Note: In the example code fragments that follow, error checking was left out to conserve space. Applications should always check the return code that the functions return. Control Program functions return an APIRET value. A return code of 0 indicates success. If a non-zero value is returned, an error occurred.

Suspending the Current Thread

An application can suspend a thread by using DosSleep. DosSleep suspends the execution of the calling thread for a specified time interval.

DosSleep requires one argument-the amount of time (in milliseconds) to suspend the thread. This value is rounded up to the nearest clock tick. If a time interval of 0 is specified, the thread gives up the remainder of the current time slice, enabling other ready threads of equal or higher priority to run; the calling thread will run again during its next scheduled time slice. If there is no other ready thread of equal or higher priority, DosSleep returns immediately; it does not yield to a thread of lower priority.

If there is a round-off error or if other threads in the system have higher priority, a thread might not resume execution immediately after the sleep interval.

The following DosSleep call suspends a thread for at least 5 seconds:

```
DosSleep(5000);
```

Note that the specified time interval refers to execution time (accumulated scheduled time slices), not to elapsed real time. Elapsed real time will be longer and will vary, depending on the hardware and on the number and priorities of other threads running in the system. In addition, even though the calling thread is scheduled for execution as soon as the specified time interval has elapsed, its execution could be delayed if a higher priority thread is running or if a hardware interrupt occurs.

Because the above factors usually cause the sleep interval to be longer than requested (though generally within a few clock ticks), DosSleep should not be used as a substitute for a real-time clock.

Note:

1. Elapsed real time for the asynchronous timers (started by DosAsyncTimer and DosStartTimer) will be much closer to their specified time intervals because these timers run independent of the execution of the calling thread.

2. To ensure optimal performance, do not use DosSleep in a single-thread Presentation Manager application. (Use WinStartTimer.)

Timing a Single Interval

To carry out other tasks while the timer counts an interval, an application can use DosAsyncTimer. This function sets a single-interval timer without stopping the application-the timer runs asynchronously to the calling thread, enabling the thread to perform other operations while it is waiting for the specified time interval to expire. When the interval elapses, OS/2 notifies the application of the expiration of the timer by posting an event semaphore. The application resets the semaphore before starting the timer and monitors the semaphore to determine when the time has elapsed. The application can use DosCreateEventSem with the initial state FALSE to create a reset semaphore. For more information on semaphores, see Semaphores.

The following code fragment creates an event semaphore and then calls DosAsyncTimer to count an interval while the application performs other tasks:

```
#define INCL_DOSDATETIME    /* Date/Time and Timer Support */
```

```
#define INCL_DOSSEMAPHORES  /* for semaphores             */

#include <os2.h>

HEV hev;
HTIMER hTimer;

/* First create a private, reset, event semaphore. */

DosCreateEventSem((PSZ) NULL,
                  &hev,
                  DC_SEM_SHARED,
                  FALSE);

/* Start async (one-shot) timer; post semaphore in 10 seconds.  */

DosAsyncTimer(10000,       /* Time in milliseconds (10 sec)     */
              (HSEM) hev,  /* Semaphore handle                  */
              &hTimer);    /* Timer handle (used to stop timer) */

    .
    .    /* Do other processing here, then wait for semaphore.  */
    .

DosWaitEventSem(hev,
                SEM_INDEFINITE_WAIT);
```

Before starting the timer, the thread creates the event semaphore with DosCreateEventSem, specifying its initial state as reset. If the semaphore was previously created by the same process, the thread resets it by calling DosResetEventSem. If the semaphore was previously created by another process, then the thread must call DosOpenEventSem to gain access to the semaphore before calling DosResetEventSem.

Next, the thread calls DosAsyncTimer, specifying the handle of the event semaphore and the desired time interval. The thread can then perform other tasks. However, in order for the application to be notified of the expiration of the timer, one or more threads in the application must call DosWaitEventSem.

When the time interval expires, the system posts the semaphore, and any threads that were blocked on DosWaitEventSem requests can resume their execution. If another time interval is required, the semaphore is reset, and both DosAsyncTimer and DosWaitEventSem are called again. (To time regular repeated intervals, use DosStartTimer.)

The timer can be canceled before its time interval expires by calling DosStopTimer.

Timing Repeated Intervals

To count an interval repeatedly, an application can use DosStartTimer. This function starts a repeated-interval timer.

Unlike DosAsyncTimer, DosStartTimer does not stop after the first interval is counted. The timer runs asynchronously to the calling thread, enabling the thread to perform other operations while it is waiting for the specified time intervals to expire. The system notifies the application of the timer's expirations by posting an event semaphore.

The application resets the semaphore before starting the timer and whenever the system posts the semaphore. The application can use the value returned in the post-counting parameter by DosResetEventSem to assure the semaphore was posted only once before it was reset.

The following code fragment starts a timer and then waits on and resets the associated event semaphore. Assume that the handle of the targeted event semaphore has been placed into *SemHandle* .

```
#define INCL_DOSDATETIME   /* Date and time values */
#include <os2.h>
#include <stdio.h>

ULONG    ulTimeInterval;   /* Interval (milliseconds) */
HSEM     hsemSemHandle;    /* Event-semaphore handle  */
HTIMER   hHandle;          /* Timer handle (returned) */
APIRET   ulrc;             /* Return code             */

ulTimeInterval = 30000;    /* Set the periodic time interval to */
                           /* elapse every 30 seconds           */

ulrc = DosStartTimer(ulTimeInterval,
                     hsemSemHandle,
                     &hHandle);
```

```
if (ulrc != 0) {
    printf("DosStartTimer error: return code = %ld",
           ulrc);
    return;
}
```

On successful return, *Handle* will contain the handle of this periodic timer. DosStopTimer can be used later to stop the periodic timer.

A repeated timer will continue to count the interval and post the semaphore until the application terminates or the application uses DosStopTimer to stop the timer explicitly. The following code fragment shows how to stop a periodic timer that has been started previously with DosStartTimer:

```
#define INCL_DOSDATETIME   /* Date and time values */
#include <os2.h>
#include <stdio.h>

HTIMER   hHandle;     /* Handle of the timer */
APIRET   ulrc;        /* Return code         */

ulrc = DosStopTimer(hHandle);

if (ulrc != 0) {
    printf("DosStopTimer error: return code = %ld",
           ulrc);
    return;
}
```

Before starting the timer, the event semaphore must be reset. If the semaphore does not exist, the thread can create it with DosCreateEventSem, specifying its initial state as reset. If the semaphore was previously created by the same process, the thread resets it by calling DosResetEventSem. If the semaphore was previously created by another process, then the thread calls DosOpenEventSem to gain access to the semaphore before calling DosResetEventSem.

Next, the thread calls DosStartTimer, specifying the handle of the event semaphore and the desired time interval. The thread can then perform other tasks. However, in order for the application to be notified of the timer's expirations, one or more threads in the application must call DosWaitEventSem.

When the time interval expires, the system posts the semaphore, and any threads that were blocked on DosWaitEventSem requests can resume their execution. Each time the semaphore is posted, it must be reset with DosResetEventSem before the next timer expiration. DosWaitEventSem must also be called to wait for the semaphore to be posted again.

In addition to resetting the event semaphore, DosResetEventSem returns the semaphore's post count (the number of times the semaphore has been posted since the last time it was in the set state). An application can use the post count to ensure that it has not missed a timer interval; if the post count is greater than one, the application missed a timer interval.

Glossary

This glossary defines many of the terms used in this book. It includes terms and definitions from the *IBM Dictionary of Computing* , as well as terms specific to the OS/2 operating system and the Presentation Manager. It is not a complete glossary for the entire OS/2 operating system; nor is it a complete dictionary of computer terms.

Other primary sources for these definitions are:

- The *American National Standard Dictionary for Information Systems* , ANSI X3.172-1990, copyrighted 1990 by the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. These definitions are identified by the symbol (A) after the definition.
- The *Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

Glossary Listing

Select a starting letter of glossary terms:

A N
B O
C P
D Q
E R
F S
G T
H U
I V
J W
K X
L Y
M Z

Glossary - A

accelerator - In SAA Common User Access architecture, a key or combination of keys that invokes an application-defined function.

accelerator table - A table used to define which key strokes are treated as *accelerators* and the commands they are translated into.

access mode - The manner in which an application gains access to a file it has opened. Examples of access modes are read-only, write-only, and read/write.

access permission - All access rights that a user has regarding an object. (I)

action - One of a set of defined tasks that a computer performs. Users request the application to perform an action in several ways, such as typing a command, pressing a function key, or selecting the action name from an action bar or menu.

action bar - In SAA Common User Access architecture, the area at the top of a window that contains choices that give a user access to actions available in that window.

action point - The current position on the screen at which the pointer is pointing. Contrast with *hot spot* and *input focus* .

active program - A program currently running on the computer. An active program can be interactive (running and receiving input from the user) or noninteractive (running but not receiving input from the user). See also *interactive program* and *noninteractive program* .

active window - The window with which the user is currently interacting.

address space - (1) The range of addresses available to a program. (A) (2) The area of virtual storage available for a particular job.

alphanumeric video output - Output to the logical video buffer when the video adapter is in text mode and the logical video buffer is addressed by an application as a rectangular array of character cells.

American National Standard Code for Information Interchange - The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

Note: IBM has defined an extension to ASCII code (characters 128-255).

anchor - A window procedure that handles Presentation Manager message conversions between an icon procedure and an application.

anchor block - An area of Presentation-Manager-internal resources to allocated process or thread that calls WinInitialize.

anchor point - A point in a window used by a program designer or by a window manager to position a subsequently appearing window.

ANSI - American National Standards Institute.

APA - All points addressable.

API - Application programming interface.

application - A collection of software components used to perform specific types of work on a computer; for example, a payroll application, an airline reservation application, a network application.

application object - In SAA Advanced Common User Access architecture, a form that an application provides for a user; for example, a

spreadsheet form. Contrast with *user object* .

application programming interface (API) - A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application-modal - Pertaining to a message box or dialog box for which processing must be completed before further interaction with any other window owned by the same application may take place.

area - In computer graphics, a filled shape such as a solid rectangle.

ASCII - American National Standard Code for Information Interchange.

ASCIIZ - A string of ASCII characters that is terminated with a byte containing the value 0.

aspect ratio - In computer graphics, the width-to-height ratio of an area, symbol, or shape.

asynchronous (ASYNC) - (1) Pertaining to two or more processes that do not depend upon the occurrence of specific events such as common timing signals. (T) (2) Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions. See also *synchronous* .

atom - A constant that represents a string. As soon as a string has been defined as an atom, the atom can be used in place of the string to save space. Strings are associated with their respective atoms in an *atom table* . See also *integer atom* .

atom table - A table used to relate *atoms* with the strings that they represent. Also in the table is the mechanism by which the presence of a string can be checked.

atomic operation - An operation that completes its work on an object before another operation can be performed on the same object.

attribute - A characteristic or property that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *graphics attributes* and *segment attributes* .

automatic link - In Information Presentation Facility (IPF), a link that begins a chain reaction at the primary window. When the user selects the primary window, an automatic link is activated to display secondary windows.

AVIO - Advanced Video Input/Output.

Glossary - B

Bézier curve - (1) A mathematical technique of specifying smooth continuous lines and surfaces, which require a starting point and a finishing point with several intermediate points that influence or control the path of the linking curve. Named after Dr. P. Bézier. (2) (D of C) In the AIX Graphics Library, a cubic spline approximation to a set of four control points that passes through the first and fourth control points and that has a continuous slope where two spline segments meet. Named after Dr. P. Bézier.

background - (1) In multiprogramming, the conditions under which low-priority programs are executed. Contrast with *foreground* . (2) An active session that is not currently displayed on the screen.

background color - The color in which the background of a graphic primitive is drawn.

background mix - An attribute that determines how the background of a graphic primitive is combined with the existing color of the graphics presentation space. Contrast with *mix* .

background program - In multiprogramming, a program that executes with a low priority. Contrast with *foreground program* .

bit map - A representation in memory of the data displayed on an APA device, usually the screen.

block - (1) A string of data elements recorded or transmitted as a unit. The elements may be characters, words, or logical records. (T) (2) To record data in a block. (3) A collection of contiguous records recorded as a unit. Blocks are separated by interblock gaps and each block may contain one or more records. (A)

block device - A storage device that performs I/O operations on blocks of data called *sectors* . Data on block devices can be randomly accessed. Block devices are designated by a drive letter (for example, **C:**).

blocking mode - A condition set by an application that determines when its threads might block. For example, an application might set the Pipemode parameter for the DosCreateNPipe function so that its threads perform I/O operations to the named pipe block when no data is available.

border - A visual indication (for example, a separator line or a background color) of the boundaries of a window.

boundary determination - An operation used to compute the size of the smallest rectangle that encloses a graphics object on the screen.

breakpoint - (1) A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. (T) (2) A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broken pipe - When all of the handles that access one end of a pipe have been closed.

bucket - One or more fields in which the result of an operation is kept.

buffer - (1) A portion of storage used to hold input or output data temporarily. (2) To allocate and schedule the use of buffers. (A)

button - A mechanism used to request or initiate an action. See also *barrel buttons* , *bezel buttons* , *mouse button* , *push button* , and *radio button* .

byte pipe - Pipes that handle data as byte streams. All unnamed pipes are byte pipes. Named pipes can be byte pipes or message pipes. See *byte stream* .

byte stream - Data that consists of an unbroken stream of bytes.

Glossary - C

cache - A high-speed buffer storage that contains frequently accessed instructions and data; it is used to reduce access time.

cached micro presentation space - A presentation space from a Presentation-Manager-owned store of micro presentation spaces. It can be used for drawing to a window only, and must be returned to the store when the task is complete.

CAD - Computer-Aided Design.

call - (1) The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (I) (A) (2) To transfer control to a procedure, program, routine, or subroutine.

calling sequence - A sequence of instructions together with any associated data necessary to execute a call. (T)

Cancel - An action that removes the current window or menu without processing it, and returns the previous window.

cascaded menu - In the OS/2 operating system, a menu that appears when the arrow to the right of a cascading choice is selected. It contains a set of choices that are related to the cascading choice. Cascaded menus are used to reduce the length of a menu. See also *cascading choice* .

cascading choice - In SAA Common User Access architecture, a choice in a menu that, when selected, produces a cascaded menu containing other choices. An arrow () appears to the right of the cascading choice.

CASE statement - In PM programming, provides the body of a window procedure. There is usually one CASE statement for each message type supported by an application.

CGA - Color graphics adapter.

chained list - A list in which the data elements may be dispersed but in which each data element contains information for locating the next. (T) Synonymous with *linked list* .

character - A letter, digit, or other symbol.

character box - In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single character from a character set. See also *character mode* . Contrast with *character cell* .

character cell - The physical, rectangular space in which any single character is displayed on a screen or printer device. Position is addressed by row and column coordinates. Contrast with *character box* .

character code - The means of addressing a character in a character set, sometimes called *code point* .

character device - A device that performs I/O operations on one character at a time. Because character devices view data as a stream of bytes, character-device data cannot be randomly accessed. Character devices include the keyboard, mouse, and printer, and are referred to by name.

character mode - A mode that, in conjunction with the font type, determines the extent to which graphics characters are affected by the character box, shear, and angle attributes.

character set - (1) An ordered set of unique representations called characters; for example, the 26 letters of English alphabet, Boolean 0 and

1, the set of symbols in the Morse code, and the 128 ASCII characters. (A) (2) All the valid characters for a programming language or for a computer system. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print.

check box - In SAA Advanced Common User Access architecture, a square box with associated text that represents a choice. When a user selects a choice, an X appears in the check box to indicate that the choice is in effect. The user can clear the check box by selecting the choice again. Contrast with *radio button* .

check mark - (1) (D of C) In SAA Advanced Common User Access architecture, a symbol that shows that a choice is currently in effect. (2) The symbol that is used to indicate a selected item on a pull-down menu.

child process - In the OS/2 operating system, a process started by another process, which is called the parent process. Contrast with *parent process* .

child window - A window that appears within the border of its parent window (either a primary window or another child window). When the parent window is resized, moved, or destroyed, the child window also is resized, moved, or destroyed; however, the child window can be moved or resized independently from the parent window, within the boundaries of the parent window. Contrast with *parent window* .

choice - (1) An option that can be selected. The choice can be presented as text, as a symbol (number or letter), or as an icon (a pictorial symbol). (2) (D of C) In SAA Common User Access architecture, an item that a user can select.

chord - (1) To press more than one button on a pointing device while the pointer is within the limits that the user has specified for the operating environment. (2) (D of C) In graphics, a short line segment whose end points lie on a circle. Chords are a means for producing a circular image from straight lines. The higher the number of chords per circle, the smoother the circular image.

class - A way of categorizing objects based on their behavior and shape. A class is, in effect, a definition of a generic object. In SOM, a class is a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior (more precisely, all of the objects manufactured by a class have the same memory layout and share a common set of methods). New classes can be defined in terms of existing classes through a technique known as *inheritance* .

class method - A class method of class <X> is a method provided by the metaclass of class <X>. Class methods are executed without requiring any instances of class <X> to exist, and are frequently used to create instances. In System Object Model, an action that can be performed on a class object.

class object - In System Object Model, the run-time implementation of a class.

class style - The set of properties that apply to every window in a window class.

client - (1) A functional unit that receives shared services from a server. (T) (2) A user, as in a client process that uses a named pipe or queue that is created and owned by a server process.

client area - The part of the window, inside the border, that is below the menu bar. It is the user's work space, where a user types information and selects choices from selection fields. In primary windows, it is where an application programmer presents the objects that a user works on.

client program - An application that creates and manipulates instances of classes.

client window - The window in which the application displays output and receives input. This window is located inside the frame window, under the window title bar and any menu bar, and within any scroll bars.

clip limits - The area of the paper that can be reached by a printer or plotter.

clipboard - In SAA Common User Access architecture, an area of computer memory, or storage, that temporarily holds data. Data in the clipboard is available to other applications.

clipping - In computer graphics, removing those parts of a display image that lie outside a given boundary. (I) (A)

clipping area - The area in which the window can paint.

clipping path - A clipping boundary in world-coordinate space.

clock tick - The minimum unit of time that the system tracks. If the system timer currently counts at a rate of X Hz, the system tracks the time every 1/X of a second. Also known as *time tick* .

CLOCK$ - Character-device name reserved for the system clock.

code page - An assignment of graphic characters and control-function meanings to all code points.

code point - (1) Synonym for *character code* . (2) (D of C) A 1-byte code representing one of 256 potential characters.

code segment - An executable section of programming code within a load module.

color dithering - See *dithering* .

color graphics adapter (CGA) - An adapter that simultaneously provides four colors and is supported by all IBM Personal Computer and Personal System/2 models.

command - The name and parameters associated with an action that a program can perform.

command area - An area composed of a command field prompt and a command entry field.

command entry field - An entry field in which users type commands.

command line - On a display screen, a display line, sometimes at the bottom of the screen, in which only commands can be entered.

command mode - A state of a system or device in which the user can enter commands.

command prompt - A field prompt showing the location of the command entry field in a panel.

Common Programming Interface (CPI) - Definitions of those application development languages and services that have, or are intended to have, implementations on and a high degree of commonality across the SAA environments. One of the three SAA architectural areas. See also *Common User Access architecture* .

Common User Access (CUA) architecture - Guidelines for the dialog between a human and a workstation or terminal. One of the three SAA architectural areas. See also *Common Programming Interface* .

compile - To translate a program written in a higher-level programming language into a machine language program.

composite window - A window composed of other windows (such as a frame window, frame-control windows, and a client window) that are kept together as a unit and that interact with each other.

computer-aided design (CAD) - The use of a computer to design or change a product, tool, or machine, such as using a computer for drafting or illustrating.

COM1, COM2, COM3 - Character-device names reserved for serial ports 1 through 3.

CON - Character-device name reserved for the console keyboard and screen.

conditional cascaded menu - A pull-down menu associated with a menu item that has a cascade mini-push button beside it in an object's pop-up menu. The conditional cascaded menu is displayed when the user selects the mini-push button.

container - In SAA Common User Access architecture, an object that holds other objects. A folder is an example of a container object. See also *folder* and *object* .

contextual help - In SAA Common User Access Architecture, help that gives specific information about the item the cursor is on. The help is contextual because it provides information about a specific item as it is currently being used. Contrast with *extended help* .

contiguous - Touching or joining at a common edge or boundary, for example, an unbroken consecutive series of storage locations.

control - In SAA Advanced Common User Access architecture, a component of the user interface that allows a user to select choices or type information; for example, a check box, an entry field, a radio button.

control area - A storage area used by a computer program to hold control information. (I) (A)

Control Panel - In the Presentation Manager, a program used to set up user preferences that act globally across the system.

Control Program - (1) The basic functions of the operating system, including DOS emulation and the support for keyboard, mouse, and video input/output. (2) A computer program designed to schedule and to supervise the execution of programs of a computer system. (I) (A)

control window - A window that is used as part of a composite window to perform simple input and output tasks. Radio buttons and check boxes are examples.

control word - An instruction within a document that identifies its parts or indicates how to format the document.

coordinate space - A two-dimensional set of points used to generate output on a video display of printer.

Copy - A choice that places onto the clipboard, a copy of what the user has selected. See also *Cut* and *Paste* .

correlation - The action of determining which element or object within a picture is at a given position on the display. This follows a *pick* operation.

coverpage window - A window in which the application's help information is displayed.

CPI - Common Programming Interface.

critical extended attribute - An extended attribute that is necessary for the correct operation of the system or a particular application.

critical section - (1) In programming languages, a part of an asynchronous procedure that cannot be executed simultaneously with a certain part of another asynchronous procedure. (I)

Note: Part of the other asynchronous procedure also is a critical section. (2) A section of code that is not reentrant; that is, code that can be executed by only one thread at a time.

CUA architecture - Common User Access architecture.

current position - In computer graphics, the position, in user coordinates, that becomes the starting point for the next graphics routine, if that routine does not explicitly specify a starting point.

cursor - A symbol displayed on the screen and associated with an input device. The cursor indicates where input from the device will be placed. Types of cursors include text cursors, graphics cursors, and selection cursors. Contrast with *pointer* and *input focus* .

Cut - In SAA Common User Access architecture, a choice that removes a selected object, or a part of an object, to the clipboard, usually compressing the space it occupied in a window. See also *Copy* and *Paste* .

Glossary - D

daisy chain - A method of device interconnection for determining interrupt priority by connecting the interrupt sources serially.

data segment - A nonexecutable section of a program module; that is, a section of a program that contains data definitions.

data structure - The syntactic structure of symbolic expressions and their storage-allocation characteristics. (T)

data transfer - The movement of data from one object to another by way of the clipboard or by direct manipulation.

DBCS - Double-byte character set.

DDE - Dynamic data exchange.

deadlock - (1) Unresolved contention for the use of a resource. (2) An error condition in which processing cannot continue because each of two elements of the process is waiting for an action by, or a response from, the other. (3) An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state.

debug - To detect, diagnose, and eliminate errors in programs. (T)

decipoint - In printing, one tenth of a point. There are 72 points in an inch.

default procedure - A function provided by the Presentation Manager Interface that may be used to process standard messages from dialogs or windows.

default value - A value assumed when no value has been specified. Synonymous with assumed value. For example, in the graphics programming interface, the default line-type is 'solid'.

definition list - A type of list that pairs a term and its description.

delta - An application-defined threshold, or number of container items, from either end of the list.

descendant - See *child process* .

descriptive text - Text used in addition to a field prompt to give more information about a field.

Deselect all - A choice that cancels the selection of all of the objects that have been selected in that window.

Desktop Manager - In the Presentation Manager, a window that displays a list of groups of programs, each of which can be started or stopped.

desktop window - The window, corresponding to the physical device, against which all other types of windows are established.

detached process - A background process that runs independent of the parent process.

detent - A point on a slider that represents an exact value to which a user can move the slider arm.

device context - A logical description of a data destination such as memory, metafile, display, printer, or plotter. See also *direct device context* , *information device context* , *memory device context* , *metafile device context* , *queued device context* , and *screen device context* .

device driver - A file that contains the code needed to attach and use a device such as a display, printer, or plotter.

device space - (1) Coordinate space in which graphics are assembled after all GPI transformations have been applied. Device space is defined in device-specific units. (2) (D of C) In computer graphics, a space defined by the complete set of addressable points of a display device. (A)

dialog - The interchange of information between a computer and its user through a sequence of requests by the user and the presentation of responses by the computer.

dialog box - In SAA Advanced Common User Access architecture, a movable window, fixed in size, containing controls that a user uses to provide information required by an application so that it can continue to process a user request. See also *message box, primary window, secondary window* . Also known as a *pop-up window* .

Dialog Box Editor - A *WYSIWYG* editor that creates dialog boxes for communicating with the application user.

dialog item - A component (for example, a menu or a button) of a dialog box. Dialog items are also used when creating dialog templates.

dialog procedure - A dialog window that is controlled by a window procedure. It is responsible for responding to all messages sent to the dialog window.

dialog tag language - A markup language used by the DTL compiler to create dialog objects.

dialog template - The definition of a dialog box, which contains details of its position, appearance, and window ID, and the window ID of each of its child windows.

direct device context - A logical description of a data destination that is a device other than the screen (for example, a printer or plotter), and where the output is not to go through the spooler. Its purpose is to satisfy queries. See also *device context* .

direct manipulation - The user's ability to interact with an object by using the mouse, typically by dragging an object around on the Desktop and dropping it on other objects.

direct memory access (DMA) - A technique for moving data directly between main storage and peripheral equipment without requiring processing of the data by the processing unit.(T)

directory - A type of file containing the names and controlling information for other files or other directories.

display point - Synonym for *pel* .

dithering - (1) The process used in color displays whereby every other pel is set to one color, and the intermediate pels are set to another. Together they produce the effect of a third color at normal viewing distances. This process can only be used on solid areas of color; it does not work, for example, on narrow lines. (2) (D of C) In computer graphics, a technique of interleaving dark and light pixels so that the resulting image looks smoothly shaded when viewed from a distance.

DMA - Direct memory access.

DOS Protect Mode Interface (DPMI) - An interface between protect mode and real mode programs.

double-byte character set (DBCS) - A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more characters than can be represented by 256 code points, require double-byte character sets. Since each character requires two bytes, the entering, displaying, and printing of DBCS characters requires hardware and software that can support DBCS.

doubleword - A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. (A)

DPMI - DOS Protect Mode Interface.

drag - In SAA Common User Access, to use a pointing device to move an object; for example, clicking on a window border, and dragging it to make the window larger.

dragging - (1) In computer graphics, moving an object on the display screen as if it were attached to the pointer. (2) (D of C) In computer graphics, moving one or more segments on a display surface by translating. (I) (A)

drawing chain - See *segment chain* .

drop - To fix the position of an object that is being dragged, by releasing the select button of the pointing device. See also *drag* .

DTL - Dialog tag language.

dual-boot function - A feature of the OS/2 operating system that allows the user to start DOS from within the operating system, or an OS/2 session from within DOS.

duplex - Pertaining to communication in which data can be sent and received at the same time. Synonymous with *full duplex* .

dynamic data exchange (DDE) - A message protocol used to communicate between applications that share data. The protocol uses shared memory as the means of exchanging data between applications.

dynamic data formatting - A formatting procedure that enables you to incorporate text, bit maps or metafiles in an IPF window at execution time.

dynamic link library - A collection of executable programming code and data that is bound to an application at load time or run time, rather than during linking. The programming code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic linking - The process of resolving external references in a program module at load time or run time rather than during linking.

dynamic segments - Graphics segments drawn in exclusive-OR mix mode so that they can be moved from one screen position to another without affecting the rest of the displayed picture.

dynamic storage - (1) A device that stores data in a manner that permits the data to move or vary with time such that the specified data is not always available for recovery. (A) (2) A storage in which the cells require repetitive application of control signals in order to retain stored data. Such repetitive application of the control signals is called a refresh operation. A dynamic storage may use static addressing or sensing circuits. (A) (3) See also *static storage* .

dynamic time slicing - Varies the size of the time slice depending on system load and paging activity.

dynamic-link module - A module that is linked at load time or run time.

Glossary - E

EBCDIC - Extended binary-coded decimal interchange code. A coded character set consisting of 8-bit coded characters (9 bits including parity check), used for information interchange among data processing systems, data communications systems, and associated equipment.

edge-triggered - Pertaining to an event semaphore that is posted then reset before a waiting thread gets a chance to run. The semaphore is considered to be posted for the rest of that thread's waiting period; the thread does not have to wait for the semaphore to be posted again.

EGA - Extended graphics adapter.

element - An entry in a graphics segment that comprises one or more graphics orders and that is addressed by the element pointer.

EMS - Expanded Memory Specification.

encapsulation - Hiding an object's implementation, that is, its private, internal data and methods. Private variables and methods are accessible only to the object that contains them.

entry field - In SAA Common User Access architecture, an area where a user types information. Its boundaries are usually indicated. See also *selection field* .

entry panel - A defined panel type containing one or more entry fields and protected information such as headings, prompts, and explanatory text.

entry-field control - The component of a user interface that provides the means by which the application receives data entered by the user in an entry field. When it has the input focus, the entry field displays a flashing pointer at the position where the next typed character will go.

environment segment - The list of environment variables and their values for a process.

environment strings - ASCII text strings that define the value of environment variables.

environment variables - Variables that describe the execution environment of a process. These variables are named by the operating system or by the application. Environment variables named by the operating system are PATH, DPATH, INCLUDE, INIT, LIB, PROMPT, and TEMP. The values of environment variables are defined by the user in the CONFIG.SYS file, or by using the SET command at the OS/2 command prompt.

error message - An indication that an error has been detected. (A)

event semaphore - A semaphore that enables a thread to signal a waiting thread or threads that an event has occurred or that a task has been completed. The waiting threads can then perform an action that is dependent on the completion of the signaled event.

exception - An abnormal condition such as an I/O error encountered in processing a data set or a file.

exclusive system semaphore - A system semaphore that can be modified only by threads within the same process.

executable file - (1) A file that contains programs or commands that perform operations or actions to be taken. (2) A collection of related data records that execute programs.

exit - To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T) Repeated exit requests return the user to the point from which all functions provided to the system are accessible. Contrast with *cancel* .

expanded memory specification (EMS) - Enables DOS applications to access memory above the 1MB real mode addressing limit.

extended attribute - An additional piece of information about a file object, such as its data format or category. It consists of a name and a value. A file object may have more than one extended attribute associated with it.

extended help - In SAA Common User Access architecture, a help action that provides information about the contents of the application window from which a user requested help. Contrast with *contextual help* .

extended-choice selection - A mode that allows the user to select more than one item from a window. Not all windows allow extended choice selection. Contrast with *multiple-choice selection* .

extent - Continuous space on a disk or diskette that is occupied by or reserved for a particular data set, data space, or file.

external link - In Information Presentation Facility, a link that connects external online document files.

Glossary - F

family-mode application - An application program that can run in the OS/2 environment and in the DOS environment; however, it cannot take advantage of many of the OS/2-mode facilities, such as multitasking, interprocess communication, and dynamic linking.

FAT - File allocation table.

FEA - Full extended attribute.

field-level help - Information specific to the field on which the cursor is positioned. This help function is "contextual" because it provides information about a specific item as it is currently used; the information is dependent upon the context within the work session.

FIFO - First-in-first-out. (A)

file - A named set of records stored or processed as a unit. (T)

file allocation table (FAT) - In IBM personal computers, a table used by the operating system to allocate space on a disk for a file, and to locate and chain together parts of the file that may be scattered on different sectors so that the file can be used in a random or sequential manner.

file attribute - Any of the attributes that describe the characteristics of a file.

File Manager - In the Presentation Manager, a program that displays directories and files, and allows various actions on them.

file specification - The full identifier for a file, which includes its drive designation, path, file name, and extension.

file system - The combination of software and hardware that supports storing information on a storage device.

file system driver (FSD) - A program that manages file I\O and controls the format of information on the storage media.

fillet - A curve that is tangential to the end points of two adjoining lines. See also *polyfillet* .

filtering - An application process that changes the order of data in a queue.

first-in-first-out (FIFO) - A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

flag - (1) An indicator or parameter that shows the setting of a switch. (2) A character that signals the occurrence of some condition, such as the end of a word. (A) (3) (D of C) A characteristic of a file or directory that enables it to be used in certain ways. See also *archive flag* , *hidden flag* , and *read-only flag* .

focus - See *input focus* .

folder - A container used to organize objects.

font - A particular size and style of typeface that contains definitions of character sets, marker sets, and pattern sets.

Font Editor - A utility program provided with the IBM Developers Toolkit that enables the design and creation of new fonts.

foreground program - (1) The program with which the user is currently interacting. Also known as *interactive program* . Contrast with *background program* . (2) (D of C) In multiprogramming, a high-priority program.

frame - The part of a window that can contain several different visual elements specified by the application, but drawn and controlled by the Presentation Manager. The frame encloses the client area.

frame styles - Standard window layouts provided by the Presentation Manager.

FSD - File system driver.

full-duplex - Synonym for *duplex* .

full-screen application - An application that has complete control of the screen.

function - (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a call. (2) A set of related control statements that cause one or more programs to be performed.

function key - A key that causes a specified sequence of operations to be performed when it is pressed, for example, F1 and Alt-K.

function key area - The area at the bottom of a window that contains function key assignments such as F1=Help.

Glossary - G

GDT - Global Descriptor Table.

general protection fault - An exception condition that occurs when a process attempts to use storage or a module that has some level of protection assigned to it, such as I/O privilege level. See also *IOPL code segment* .

Global Descriptor Table (GDT) - A table that defines code and data segments available to all tasks in an application.

global dynamic-link module - A dynamic-link module that can be shared by all processes in the system that refer to the module name.

global file-name character - Either a question mark (?) or an asterisk (*) used as a variable in a file name or file name extension when referring to a particular file or group of files.

glyph - A graphic symbol whose appearance conveys information.

GPI - Graphics programming interface.

graphic primitive - In computer graphics, a basic element, such as an arc or a line, that is not made up of smaller parts and that is used to create diagrams and pictures. See also *graphics segment* .

graphics - (1) A picture defined in terms of graphic primitives and graphics attributes. (2) (D of C) The making of charts and pictures. (3) Pertaining to charts, tables, and their creation. (4) See *computer graphics, coordinate graphics, fixed-image graphics, interactive graphics, passive graphics, raster graphics* .

graphics attributes - Attributes that apply to graphic primitives. Examples are color, line type, and shading-pattern definition. See also *segment attributes* .

graphics field - The clipping boundary that defines the visible part of the presentation-page contents.

graphics mode - One of several states of a display. The mode determines the resolution and color content of the screen.

graphics model space - The conceptual coordinate space in which a picture is constructed after any model transforms have been applied. Also known as *model space* .

Graphics programming interface - The formally defined programming language that is between an IBM graphics program and the user of the program.

graphics segment - A sequence of related graphic primitives and graphics attributes. See also *graphic primitive* .

graying - The indication that a choice on a pull-down is unavailable.

group - A collection of logically connected controls. For example, the buttons controlling paper size for a printer could be called a group. See also *program group* .

Glossary - H

handle - (1) An identifier that represents an object, such as a device or window, to the Presentation Interface. (2) (D of C) In the Advanced DOS and OS/2 operating systems, a binary value created by the system that identifies a drive, directory, and file so that the file can be found and opened.

hard error - An error condition on a network that requires either that the system be reconfigured or that the source of the error be removed before the system can resume reliable operation.

header - (1) System-defined control information that precedes user data. (2) The portion of a message that contains control information for the message, such as one or more destination fields, name of the originating station, input sequence number, character string indicating the type of message, and priority level for the message.

heading tags - A document element that enables information to be displayed in windows, and that controls entries in the contents window controls placement of push buttons in a window, and defines the shape and size of windows.

heap - An area of free storage available for dynamic allocation by an application. Its size varies according to the storage requirements of the application.

help function - (1) A function that provides information about a specific field, an application panel, or information about the help facility. (2) (D of C) One or more display images that describe how to use application software or how to do a system operation.

Help index - In SAA Common User Access architecture, a help action that provides an index of the help information available for an application.

help panel - A panel with information to assist users that is displayed in response to a help request from the user.

help window - A Common-User-Access-defined secondary window that displays information when the user requests help.

hidden file - An operating system file that is not displayed by a directory listing.

hide button - In the OS/2 operating system, a small, square button located in the right-hand corner of the title bar of a window that, when selected, removes from the screen all the windows associated with that window. Contrast with *maximize button* . See also *restore button* .

hierarchical inheritance - The relationship between parent and child classes. An object that is lower in the inheritance hierarchy than another object, inherits all the characteristics and behaviors of the objects above it in the hierarchy.

hierarchy - A tree of segments beginning with the root segment and proceeding downward to dependent segment types.

high-performance file system (HPFS) - In the OS/2 operating system, an installable file system that uses high-speed buffer storage, known as a cache, to provide fast access to large disk volumes. The file system also supports the coexistence of multiple, active file systems on a single personal computer, with the capability of multiple and different storage devices. File names used with the HPFS can have as many as 254 characters.

hit testing - The means of identifying which window is associated with which input device event.

hook - A point in a system-defined function where an application can supply additional code that the system processes as though it were part of the function.

hook chain - A sequence of hook procedures that are "chained" together so that each event is passed, in turn, to each procedure in the chain.

hot spot - The part of the pointer that must touch an object before it can be selected. This is usually the tip of the pointer. Contrast with *action point* .

HPFS - high-performance file system.

hypergraphic link - A connection between one piece of information and another through the use of graphics.

hypertext - A way of presenting information online with connections between one piece of information and another, called *hypertext links* . See also *hypertext link* .

hypertext link - A connection between one piece of information and another.

Glossary - I

I/O operation - An input operation to, or output operation from a device attached to a computer.

I-beam pointer - A pointer that indicates an area, such as an entry field in which text can be edited.

icon - In SAA Advanced Common User Access architecture, a graphical representation of an object, consisting of an image, image

background, and a label. Icons can represent items (such as a document file) that the user wants to work on, and actions that the user wants to perform. In the Presentation Manager, icons are used for data objects, system actions, and minimized programs.

icon area - In the Presentation Manager, the area at the bottom of the screen that is normally used to display the icons for minimized programs.

Icon Editor - The Presentation Manager-provided tool for creating icons.

IDL - Interface Definition Language.

image font - A set of symbols, each of which is described in a rectangular array of pels. Some of the pels in the array are set to produce the image of one of the symbols. Contrast with *outline font* .

implied metaclass - Subclassing the metaclass of a parent class without a separate CSC for the resultant metaclass.

indirect manipulation - Interaction with an object through choices and controls.

information device context - A logical description of a data destination other than the screen (for example, a printer or plotter), but where no output will occur. Its purpose is to satisfy queries. See also *device context* .

information panel - A defined panel type characterized by a body containing only protected information.

Information Presentation Facility (IPF) - A facility provided by the OS/2 operating system, by which application developers can produce online documentation and context-sensitive online help panels for their applications.

inheritance - The technique of specifying the shape and behavior of one class (called a *subclass*) as incremental differences from another class (called the *parent class* or *superclass*). The subclass inherits the superclass' state representation and methods, and can provide additional data elements and methods. The subclass also can provide new functions with the same method names used by the superclass. Such a subclass method is said to override the superclass method, and will be selected automatically by method resolution on subclass instances. An overriding method can elect to call upon the superclass' method as part of its own implementation.

input focus - (1) The area of a window where user interaction is possible using an input device, such as a mouse or the keyboard. (2) The position in the *active window* where a user's normal interaction with the keyboard will appear.

input router - An internal OS/2 process that removes messages from the system queue.

input/output control - A device-specific command that requests a function of a device driver.

installable file system (IFS) - A file system in which software is installed when the operating system is started.

instance - (Or object instance). A specific object, as distinguished from the abstract definition of an object referred to as its class.

instance method - A method valid for a particular object.

instruction pointer - In System/38, a pointer that provides addressability for a machine interface instruction in a program.

integer atom - An *atom* that represents a predefined system constant and carries no storage overhead. For example, names of window classes provided by Presentation Manager are expressed as integer atoms.

interactive graphics - Graphics that can be moved or manipulated by a user at a terminal.

interactive program - (1) A program that is running (active) and is ready to receive (or is receiving) input from a user. (2) A running program that can receive input from the keyboard or another input device. Compare with *active program* and contrast with *noninteractive* program.

Also known as a *foreground program* .

interchange file - A file containing data that can be sent from one Presentation Manager interface application to another.

Interface Definition Language (IDL) - Language-neutral interface specification for a SOM class.

interpreter - A program that translates and executes each instruction of a high-level programming language before it translates and executes.

interprocess communication (IPC) - In the OS/2 operating system, the exchange of information between processes or threads through semaphores, pipes, queues, and shared memory.

interval timer - (1) A timer that provides program interruptions on a program-controlled basis. (2) An electronic counter that counts intervals of time under program control.

IOCtl - Input/output control.

IOPL - Input/output privilege level.

IOPL code segment - An IOPL executable section of programming code that enables an application to directly manipulate hardware interrupts and ports without replacing the device driver. See also *privilege level* .

IPC - Interprocess communication.

IPF - Information Presentation Facility.

IPF compiler - A text compiler that interpret tags in a source file and converts the information into the specified format.

IPF tag language - A markup language that provides the instructions for displaying online information.

item - A data object that can be passed in a DDE transaction.

Glossary - J

journal - A special-purpose file that is used to record changes made in the system.

Glossary - K

Kanji - A graphic character set used in Japanese ideographic alphabets.

KBD$ - Character-device name reserved for the keyboard.

kernel - The part of an operating system that performs basic functions, such as allocating hardware resources.

kerning - The design of graphics characters so that their character boxes overlap. Used to space text proportionally.

keyboard accelerator - A keystroke that generates a command message for an application.

keyboard augmentation - A function that enables a user to press a keyboard key while pressing a mouse button.

keyboard focus - A temporary attribute of a window. The window that has a keyboard focus receives all keyboard input until the focus changes to a different window.

Keys help - In SAA Common User Access architecture, a help action that provides a listing of the application keys and their assigned functions.

Glossary - L

label - In a graphics segment, an identifier of one or more elements that is used when editing the segment.

LAN - Local area network.

language support procedure - A function provided by the Presentation Manager Interface for applications that do not, or cannot (as in the case of COBOL and FORTRAN programs), provide their own dialog or window procedures.

lazy drag - See *pickup and drop* .

lazy drag set - See *pickup set* .

LDT - In the OS/2 operating system, Local Descriptor Table.

LIFO stack - A stack from which data is retrieved in last-in, first-out order.

linear address - A unique value that identifies the memory object.

linked list - Synonym for *chained list* .

list box - In SAA Advanced Common User Access architecture, a control that contains scrollable choices from which a user can select one choice.

Note: In CUA architecture, this is a programmer term. The end user term is selection list.

list button - A button labeled with an underlined down-arrow that presents a list of valid objects or choices that can be selected for that field.

list panel - A defined panel type that displays a list of items from which users can select one or more choices and then specify one or more actions to work on those choices.

load time - The point in time at which a program module is loaded into main storage for execution.

load-on-call - A function of a linkage editor that allows selected segments of the module to be disk resident while other segments are executing. Disk resident segments are loaded for execution and given control when any entry point that they contain is called.

local area network (LAN) - (1) A computer network located on a user's premises within a limited geographical area. Communication within a local area network is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation. (T)

Note: A LAN does not use store and forward techniques. (2) A network in which a set of devices are connected to one another for communication and that can be connected to a larger network.

Local Descriptor Table (LDT) - Defines code and data segments specific to a single task.

lock - A serialization mechanism by means of which a resource is restricted for use by the holder of the lock.

logical storage device - A device that the user can map to a physical (actual) device.

LPT1, LPT2, LPT3 - Character-device names reserved for parallel printers 1 through 3.

Glossary - M

main window - The window that is positioned relative to the *desktop window* .

manipulation button - The button on a pointing device a user presses to directly manipulate an object.

map - (1) A set of values having a defined correspondence with the quantities or values of another set. (I) (A) (2) To establish a set of values having a defined correspondence with the quantities or values of another set. (I)

marker box - In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single marker from a marker set.

marker symbol - A symbol centered on a point. Graphs and charts can use marker symbols to indicate the plotted points.

marquee box - The rectangle that appears during a selection technique in which a user selects objects by drawing a box around them with a pointing device.

Master Help Index - In the OS/2 operating system, an alphabetic list of help topics related to using the operating system.

maximize - To enlarge a window to its largest possible size.

media window - The part of the physical device (display, printer, or plotter) on which a picture is presented.

memory block - Part memory within a heap.

memory device context - A logical description of a data destination that is a memory bit map. See also *device context* .

memory management - A feature of the operating system for allocating, sharing, and freeing main storage.

memory object - Logical unit of memory requested by an application, which forms the granular unit of memory manipulation from the application viewpoint.

menu - In SAA Advanced Common User Access architecture, an extension of the menu bar that displays a list of choices available for a selected choice in the menu bar. After a user selects a choice in menu bar, the corresponding menu appears. Additional pop-up windows can appear from menu choices.

menu bar - In SAA Advanced Common User Access architecture, the area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus.

menu button - The button on a pointing device that a user presses to view a pop-up menu associated with an object.

message - (1) In the Presentation Manager, a packet of data used for communication between the Presentation Manager interface and Presentation Manager applications (2) In a user interface, information not requested by users but presented to users by the computer in response to a user action or internal process.

message box - (1) A dialog window predefined by the system and used as a simple interface for applications, without the necessity of creating dialog-template resources or dialog procedures. (2) (D of C) In SAA Advanced Common User Access architecture, a type of window that shows messages to users. See also *dialog box, primary window, secondary window* .

message filter - The means of selecting which messages from a specific window will be handled by the application.

message queue - A sequenced collection of messages to be read by the application.

message stream mode - A method of operation in which data is treated as a stream of messages. Contrast with *byte stream* .

metacharacter - See *global file-name character* .

metaclass - A class whose instances are all classes. In SOM, any class descended from SOMClass is a metaclass. The methods of a metaclass are sometimes called "class" methods.

metafile - A file containing a series of attributes that set color, shape and size, usually of a picture or a drawing. Using a program that can interpret these attributes, a user can view the assembled image.

metafile device context - A logical description of a data destination that is a metafile, which is used for graphics interchange. See also *device context* .

metalanguage - A language used to specify another language. For example, data types can be described using a metalanguage so as to make the descriptions independent of any one computer language.

method - One of the units that makes up the behavior of an object. A method is a combination of a function and a name, such that many different functions can have the same name. Which function the name refers to at any point in time depends on the object that is to execute the method and is the subject of method resolution.

method override - The replacement, by a child class, of the implementation of a method inherited from a parent and an ancestor class.

mickey - A unit of measurement for physical mouse motion whose value depends on the mouse device driver currently loaded.

micro presentation space - A graphics presentation space in which a restricted set of the GPI function calls is available.

minimize - To remove from the screen all windows associated with an application and replace them with an icon that represents the application.

mix - An attribute that determines how the foreground of a graphic primitive is combined with the existing color of graphics output. Also known as *foreground mix* . Contrast with *background mix* .

mixed character string - A string containing a mixture of one-byte and *Kanji* or Hangeul (two-byte) characters.

mnemonic - (1) A method of selecting an item on a pull-down by means of typing the highlighted letter in the menu item. (2) (D of C) In SAA Advanced Common User Access architecture, usually a single character, within the text of a choice, identified by an underscore beneath the character. If all characters in a choice already serve as mnemonics for other choices, another character, placed in parentheses immediately following the choice, can be used. When a user types the mnemonic for a choice, the choice is either selected or the cursor is moved to that choice.

modal dialog box - In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that requires a user to enter information before continuing to work in the application window from which it was displayed. Contrast with *modeless dialog box* . Also known as a *serial dialog box* . Contrast with *parallel dialog box* .

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

model space - See *graphics model space* .

modeless dialog box - In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that allows users to continue their dialog with the application without entering information in the dialog box. Also known as a *parallel dialog box* . Contrast with *modal dialog box* .

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

module definition file - A file that describes the code segments within a load module. For example, it indicates whether a code segment is loadable before module execution begins (preload), or loadable only when referred to at run time (load-on-call).

mouse - In SAA usage, a device that a user moves on a flat surface to position a pointer on the screen. It allows a user to select a choice o function to be performed or to perform operations on the screen, such as dragging or drawing lines from one position to another.

MOUSE$ - Character-device name reserved for a mouse.

multiple-choice selection - In SAA Basic Common User Access architecture, a type of field from which a user can select one or more choices or select none. See also *check box* . Contrast with *extended-choice selection* .

multiple-line entry field - In SAA Advanced Common User Access architecture, a control into which a user types more than one line of information. See also *single-line entry field* .

multitasking - The concurrent processing of applications or parts of applications. A running application and its data are protected from other concurrently running applications.

mutex semaphore - (Mutual exclusion semaphore). A semaphore that enables threads to serialize their access to resources. Only the thread that currently owns the mutex semaphore can gain access to the resource, thus preventing one thread from interrupting operations being performed by another.

muxwait semaphore - (Multiple wait semaphore). A semaphore that enables a thread to wait either for multiple event semaphores to be posted or for multiple mutex semaphores to be released. Alternatively, a muxwait semaphore can be set to enable a thread to wait for any ONE of the event or mutex semaphores in the muxwait semaphore's list to be posted or released.

Glossary - N

named pipe - A named buffer that provides client-to-server, server-to-client, or full duplex communication between unrelated processes. Contrast with *unnamed pipe* .

national language support (NLS) - The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, MRI, or documentation of a product.

nested list - A list that is contained within another list.

NLS - national language support.

non-8.3 file-name format - A file-naming convention in which file names can consist of up to 255 characters. See also *8.3 file-name format* .

noncritical extended attribute - An extended attribute that is not necessary for the function of an application.

nondestructive read - Reading that does not erase the data in the source location. (T)

noninteractive program - A running program that cannot receive input from the keyboard or other input device. Compare with *active* program, and contrast with *interactive program* .

nonretained graphics - Graphic primitives that are not remembered by the Presentation Manager interface when they have been drawn. Contrast with *retained graphics* .

null character (NUL) - (1) Character-device name reserved for a nonexistent (dummy) device. (2) (D of C) A control character that is used to accomplish media-fill or time-fill and that may be inserted into or removed from a sequence of characters without affecting the meaning of the sequence; however, the control of equipment or the format may be affected by this character. (I) (A)

null-terminated string - A string of (n+1) characters where the (n+1)th character is the 'null' character (0x00) Also known as 'zero-terminated' string and 'ASCIIZ' string.

Glossary - O

object - The elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way of associating specific data values with a specific set of named functions (called *methods*) for a period of time (referred to as the *lifetime* of the object). The data values of an object are referred to as its *state* . In SOM, objects are created by other objects called *classes* . The specification of what comprises the set of functions and data elements that make up an object is referred to as the *definition* of a class.

SOM objects offer a high degree of *encapsulation* . This property permits many aspects of the implementation of an object to change without affecting client programs that depend on the object's behavior.

object definition - See *class* .

object instance - See *instance* .

Object Interface Definition Language (OIDL) - Specification language used in SOM Version 1 for defining classes. Replaced by Interface Definition Language (IDL).

object window - A window that does not have a parent but which might have child windows. An object window cannot be presented on a device.

OIDL - Object Interface Definition Language.

open - To start working with a file, directory, or other object.

ordered list - Vertical arrangements of items, with each item in the list preceded by a number or letter.

outline font - A set of symbols, each of which is created as a series of lines and curves. Synonymous with *vector font* . Contrast with *image font* .

output area - An area of storage reserved for output. (A)

owner window - A window into which specific events that occur in another (owned) window are reported.

ownership - The determination of how windows communicate using messages.

owning process - The process that owns the resources that might be shared with other processes.

Glossary - P

page - (1) A 4KB segment of contiguous physical memory. (2) (D of C) A defined unit of space on a storage medium.

page viewport - A boundary in device coordinates that defines the area of the output device in which graphics are to be displayed. The presentation-page contents are transformed automatically to the page viewport in device space.

paint - (1) The action of drawing or redrawing the contents of a window. (2) In computer graphics, to shade an area of a display image; for example, with crosshatching or color.

panel - In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

panel area - An area within a panel that contains related information. The three major Common User Access-defined panel areas are the action bar, the function key area, and the panel body.

panel area separator - In SAA Basic Common User Access architecture, a solid, dashed, or blank line that provides a visual distinction between two adjacent areas of a panel.

panel body - The portion of a panel not occupied by the action bar, function key area, title or scroll bars. The panel body can contain protected information, selection fields, and entry fields. The layout and content of the panel body determine the panel type.

panel body area - See *client area* .

panel definition - A description of the contents and characteristics of a panel. A panel definition is the application developer's mechanism for predefining the format to be presented to users in a window.

panel ID - In SAA Basic Common User Access architecture, a panel identifier, located in the upper-left corner of a panel. A user can choose whether to display the panel ID.

panel title - In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

paper size - The size of paper, defined in either standard U.S. or European names (for example, A, B, A4), and measured in inches or millimeters respectively.

parallel dialog box - See *modeless dialog box* .

parameter list - A list of values that provides a means of associating addressability of data defined in a called program with data in the calling program. It contains parameter names and the order in which they are to be associated in the calling and called program.

parent class - See *inheritance* .

parent process - In the OS/2 operating system, a process that creates other processes. Contrast with *child process* .

parent window - In the OS/2 operating system, a window that creates a child window. The child window is drawn within the parent window. If the parent window is moved, resized, or destroyed, the child window also will be moved, resized, or destroyed. However, the child window can be moved and resized independently from the parent window, within the boundaries of the parent window. Contrast with *child window* .

partition - (1) A fixed-size division of storage. (2) On an IBM personal computer fixed disk, one of four possible storage areas of variable size; one may be accessed by DOS, and each of the others may be assigned to another operating system.

Paste - A choice in the Edit pull-down that a user selects to move the contents of the clipboard into a preselected location. See also *Copy* and *Cut* .

path - The route used to locate files; the storage location of a file. A fully qualified path lists the drive identifier, directory name, subdirectory name (if any), and file name with the associated extension.

PDD - Physical device driver.

peeking - An action taken by any thread in the process that owns the queue to examine queue elements without removing them.

pel - (1) The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonym for *display point* , *pixel* , and *picture element* . (2) (D of C) Picture element.

persistent object - An object whose instance data and state are preserved between system shutdown and system startup.

physical device driver (PDD) - A system interface that handles hardware interrupts and supports a set of input and output functions.

pick - To select part of a displayed object using the pointer.

pickup - To add an object or set of objects to the pickup set.

pickup and drop - A drag operation that does not require the direct manipulation button to be pressed for the duration of the drag.

pickup set - The set of objects that have been picked up as part of a pickup and drop operation.

picture chain - See *segment chain* .

picture element - (1) Synonym for *pel* . (2) (D of C) In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity. (T) . (3) The area of the finest detail that can be reproduced effectively on the recording medium.

PID - Process identification.

pipe - (1) A named or unnamed buffer used to pass data between processes. A process reads from or writes to a pipe as if the pipe were a standard-input or standard-output file. See also *named pipe* and *unnamed pipe* . (2) (D of C) To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (|).

pixel - (1) Synonym for *pel* . (2) (D of C) Picture element.

plotter - An output unit that directly produces a hardcopy record of data on a removable medium, in the form of a two-dimensional graphic representation. (T)

PM - Presentation Manager.

pointer - (1) The symbol displayed on the screen that is moved by a pointing device, such as a *mouse* . The pointer is used to point at items that users can select. Contrast with *cursor* . (2) A data element that indicates the location of another data element. (T)

POINTER$ - Character-device name reserved for a pointer device (mouse screen support).

pointing device - In SAA Advanced Common User Access architecture, an instrument, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

pointings - Pairs of x-y coordinates produced by an operator defining positions on a screen with a pointing device, such as a *mouse* .

polyfillet - A curve based on a sequence of lines. The curve is tangential to the end points of the first and last lines, and tangential also to the midpoints of all other lines. See also *fillet* .

polygon - One or more closed figures that can be drawn filled, outlined, or filled and outlined.

polyline - A sequence of adjoining lines.

polymorphism - The ability to have different implementations of the same method for two or more classes of objects.

pop - To retrieve an item from a last-in-first-out stack of items. Contrast with *push* .

pop-up menu - A menu that lists the actions that a user can perform on an object. The contents of the pop-up menu can vary depending on the context, or state, of the object.

pop-up window - (1) A window that appears on top of another window in a dialog. Each pop-up window must be completed before returning to the underlying window. (2) (D of C) In SAA Advanced Common User Access architecture, a movable window, fixed in size, in which a user provides information required by an application so that it can continue to process a user request.

presentation drivers - Special purpose I/O routines that handle field device-independent I/O requests from the PM and its applications.

Presentation Manager (PM) - The interface of the OS/2 operating system that presents, in windows a graphics-based interface to applications and files installed and running under the OS/2 operating system.

presentation page - The coordinate space in which a picture is assembled for display.

presentation space (PS) - (1) Contains the device-independent definition of a picture. (2) (D of C) The display space on a display device.

primary window - In SAA Common User Access architecture, the window in which the main interaction between the user and the application takes place. In a multiprogramming environment, each application starts in its own primary window. The primary window remains for the duration of the application, although the panel displayed will change as the user's dialog moves forward. See also *secondary window* .

primitive - In computer graphics, one of several simple functions for drawing on the screen, including, for example, the rectangle, line, ellipse, polygon, and so on.

primitive attribute - A specifiable characteristic of a graphic primitive. See *graphics attributes* .

print job - The result of sending a document or picture to be printed.

Print Manager - In the Presentation Manager, the part of the spooler that manages the spooling process. It also allows users to view print queues and to manipulate print jobs.

privilege level - A protection level imposed by the hardware architecture of the IBM personal computer. There are four privilege levels (number 0 through 3). Only certain types of programs are allowed to execute at each privilege level. See also *IOPL code segment* .

procedure call - In programming languages, a language construct for invoking execution of a procedure.

process - An instance of an executing application and the resources it is using.

program - A sequence of instructions that a computer can interpret and execute.

program details - Information about a program that is specified in the *Program Manager* window and is used when the program is started.

program group - In the Presentation Manager, several programs that can be acted upon as a single entity.

program name - The full file specification of a program. Contrast with *program title* .

program title - The name of a program as it is listed in the *Program Manager* window. Contrast with *program name* .

prompt - A displayed symbol or message that requests input from the user or gives operational information; for example, on the display screen of an IBM personal computer, the DOS A> prompt. The user must respond to the prompt in order to proceed.

protect mode - A method of program operation that limits or prevents access to certain instructions or areas of storage. Contrast with *real mode* .

protocol - A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I)

pseudocode - An artificial language used to describe computer program algorithms without using the syntax of any particular programming language. (A)

pull-down - (1) An *action bar* extension that displays a list of choices available for a selected action bar choice. After users select an action bar choice, the pull-down appears with the list of choices. Additional *pop-up windows* may appear from pull-down choices to further extend the actions available to users. (2) (D of C) In SAA Common User Access architecture, pertaining to a choice in an action bar pull-down.

push - To add an item to a last-in-first-out stack of items. Contrast with *pop* .

push button - In SAA Advanced Common User Access architecture, a rectangle with text inside. Push buttons are used in windows for actions that occur immediately when the push button is selected.

putback - To remove an object or set of objects from the lazy drag set. This has the effect of undoing the pickup operation for those objects

putdown - To drop the objects in the lazy drag set on the target object.

Glossary - Q

queue - (1) A linked list of elements waiting to be processed in FIFO order. For example, a queue may be a list of print jobs waiting to be printed. (2) (D of C) A line or list of items waiting to be processed; for example, work to be performed or messages to be displayed.

queued device context - A logical description of a data destination (for example, a printer or plotter) where the output is to go through the spooler. See also *device context* .

Glossary - R

radio button - (1) A control window, shaped like a round button on the screen, that can be in a checked or unchecked state. It is used to select a single item from a list. Contrast with *check box* . (2) In SAA Advanced Common User Access architecture, a circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which only one can be selected. The circle is partially filled when a choice is selected.

RAS - Reliability, availability, and serviceability.

raster - (1) In computer graphics, a predetermined pattern of lines that provides uniform coverage of a display space. (T) (2) The coordinate grid that divides the display area of a display device. (A)

read-only file - A file that can be read from but not written to.

real mode - A method of program operation that does not limit or prevent access to any instructions or areas of storage. The operating system loads the entire program into storage and gives the program access to all system resources. Contrast with *protect mode* .

realize - To cause the system to ensure, wherever possible, that the physical color table of a device is set to the closest possible match in the logical color table.

recursive routine - A routine that can call itself, or be called by another routine that was called by the recursive routine.

reentrant - The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference phrase - (1) A word or phrase that is emphasized in a device-dependent manner to inform the user that additional information for the word or phrase is available. (2) (D of C) In hypertext, text that is highlighted and preceded by a single-character input field used to signify the existence of a hypertext link.

reference phrase help - In SAA Common User Access architecture, highlighted words or phrases within help information that a user selects to get additional information.

refresh - To update a window, with changed information, to its current status.

region - A clipping boundary in device space.

register - A part of internal storage having a specified storage capacity and usually intended for a specific purpose. (T)

remote file system - A file-system driver that gains access to a remote system without a block device driver.

resource - The means of providing extra information used in the definition of a window. A resource can contain definitions of fonts, templates, accelerators, and mnemonics; the definitions are held in a resource file.

resource file - A file containing information used in the definition of a window. Definitions can be of fonts, templates, accelerators, and mnemonics.

restore - To return a window to its original size or position following a sizing or moving action.

retained graphics - Graphic primitives that are remembered by the Presentation Manager interface after they have been drawn. Contrast with *nonretained graphics* .

return code - (1) A value returned to a program to indicate the results of an operation requested by that program. (2) A code used to influence the execution of succeeding instructions.(A)

reverse video - (1) A form of highlighting a character, field, or cursor by reversing the color of the character, field, or cursor with its background; for example, changing a red character on a black background to a black character on a red background. (2) In SAA Basic Common User Access architecture, a screen emphasis feature that interchanges the foreground and background colors of an item.

REXX Language - Restructured Extended Executor. A procedural language that provides batch language functions along with structured programming constructs such as loops; conditional testing and subroutines.

RGB - (1) Color coding in which the brightness of the additive primary colors of light, red, green, and blue, are specified as three distinct values of white light. (2) Pertaining to a color display that accepts signals representing red, green, and blue.

roman - Relating to a type style with upright characters.

root segment - In a hierarchical database, the highest segment in the tree structure.

round-robin scheduling - A process that allows each thread to run for a specified amount of time.

run time - (1) Any instant at which the execution of a particular computer program takes place. (T) (2) The amount of time needed for the execution of a particular computer program. (T) (3) The time during which an instruction in an instruction register is decoded and performed. Synonym for *execution time* .

Glossary - S

SAA - Systems Application Architecture.

SBCS - Single-byte character set.

scheduler - A computer program designed to perform functions such as scheduling, initiation, and termination of jobs.

screen - In SAA Basic Common User Access architecture, the physical surface of a display device upon which information is shown to a user.

screen device context - A logical description of a data destination that is a particular window on the screen. See also *device context* .

SCREEN$ - Character-device name reserved for the display screen.

scroll bar - In SAA Advanced Common User Access architecture, a part of a window, associated with a scrollable area, that a user interacts with to see information that is not currently allows visible.

scrollable entry field - An entry field larger than the visible field.

scrollable selection field - A selection field that contains more choices than are visible.

scrolling - Moving a display image vertically or horizontally in a manner such that new data appears at one edge, as existing data disappears at the opposite edge.

secondary window - A window that contains information that is dependent on information in a primary window and is used to supplement the interaction in the primary window.

sector - On disk or diskette storage, an addressable subdivision of a track used to record one block of a program or data.

segment - See *graphics segment* .

segment attributes - Attributes that apply to the segment as an entity, as opposed to the individual primitives within the segment. For example, the visibility or detectability of a segment.

segment chain - All segments in a graphics presentation space that are defined with the 'chained' attribute. Synonym for *picture chain* .

segment priority - The order in which segments are drawn.

segment store - An area in a normal graphics presentation space where retained graphics segments are stored.

select - To mark or choose an item. Note that *select* means to mark or type in a choice on the screen; *enter* means to send all selected choices to the computer for processing.

select button - The button on a pointing device, such as a mouse, that is pressed to select a menu choice. Also known as button 1.

selection cursor - In SAA Advanced Common User Access architecture, a visual indication that a user has selected a choice. It is represented by outlining the choice with a dotted box. See also *text cursor* .

selection field - (1) In SAA Advanced Common User Access architecture, a set of related choices. See also *entry field*. (2) In SAA Basic Common User Access architecture, an area of a panel that cannot be scrolled and contains a fixed number of choices.

semantics - The relationships between symbols and their meanings.

semaphore - An object used by applications for signalling purposes and for controlling access to serially reusable resources.

separator - In SAA Advanced Common User Access architecture, a line or color boundary that provides a visual distinction between two

adjacent areas.

serial dialog box - See *modal dialog box* .

serialization - The consecutive ordering of items.

serialize - To ensure that one or more events occur in a specified sequence.

serially reusable resource (SRR) - A logical resource or object that can be accessed by only one task at a time.

session - (1) A routing mechanism for user interaction via the console; a complete environment that determines how an application runs and how users interact with the application. OS/2 can manage more than one session at a time, and more than one process can run in a session. Each session has its own set of environment variables that determine where OS/2 looks for dynamic-link libraries and other important files. (2) (D of C) In the OS/2 operating system, one instance of a started program or command prompt. Each session is separate from all other sessions that might be running on the computer. The operating system is responsible for coordinating the resources that each session uses, such as computer memory, allocation of processor time, and windows on the screen.

Settings Notebook - A control window that is used to display the settings for an object and to enable the user to change them.

shadow - An object that refers to another object. A shadow is not a copy of another object, but is another representation of the object.

shadow box - The area on the screen that follows mouse movements and shows what shape the window will take if the mouse button is released.

shared data - Data that is used by two or more programs.

shared memory - In the OS/2 operating system, a segment that can be used by more than one program.

shear - In computer graphics, the forward or backward slant of a graphics symbol or string of such symbols relative to a line perpendicular to the baseline of the symbol.

shell - (1) A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens, and communicate them to the operating system. (2) Software that allows a kernel program to run under different operating-system environments.

shutdown - The process of ending operation of a system or a subsystem, following a defined procedure.

sibling processes - Child processes that have the same parent process.

sibling windows - Child windows that have the same parent window.

simple list - A list of like values; for example, a list of user names. Contrast with *mixed list* .

single-byte character set (SBCS) - A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set* .

slider box - In SAA Advanced Common User Access architecture: a part of the scroll bar that shows the position and size of the visible information in a window relative to the total amount of information available. Also known as *thumb mark* .

SOM - System Object Model.

source file - A file that contains source statements for items such as high-level language programs and data description specifications.

source statement - A statement written in a programming language.

specific dynamic-link module - A dynamic-link module created for the exclusive use of an application.

spin button - In SAA Advanced Common User Access architecture, a type of entry field that shows a scrollable ring of choices from which a user can select a choice. After the last choice is displayed, the first choice is displayed again. A user can also type a choice from the scrollable ring into the entry field without interacting with the spin button.

spline - A sequence of one or more Bézier curves.

spooler - A program that intercepts the data going to printer devices and writes it to disk. The data is printed or plotted when it is complete and the required device is available. The spooler prevents output from different sources from being intermixed.

stack - A list constructed and maintained so that the next data element to be retrieved is the most recently stored. This method is characterized as last-in-first-out (LIFO).

standard window - A collection of window elements that form a panel. The standard window can include one or more of the following window elements: sizing borders, system menu icon, title bar, maximize/minimize/restore icons, action bar and pull-downs, scroll bars, and client area.

static control - The means by which the application presents descriptive information (for example, headings and descriptors) to the user. The user cannot change this information.

static storage - (1) A read/write storage unit in which data is retained in the absence of control signals. (A) Static storage may use dynamic addressing or sensing circuits. (2) Storage other than *dynamic storage* . (A)

style - See *window style* .

subclass - A class that inherits from another class. See also *Inheritance* .

subdirectory - In an IBM personal computer, a file referred to in a root directory that contains the names of other files stored on the diskette or fixed disk.

superclass - A class from which another class inherits. See also *inheritance* .

swapping - (1) A process that interchanges the contents of an area of real storage with the contents of an area in auxiliary storage. (I) (A) (2) In a system with virtual storage, a paging technique that writes the active pages of a job to auxiliary storage and reads pages of another job from auxiliary storage into real storage. (3) The process of temporarily removing an active job from main storage, saving it on disk, and processing another job in the area of main storage formerly occupied by the first job.

switch - (1) In SAA usage, to move the cursor from one point of interest to another; for example, to move from one screen or window to another or from a place within a displayed image to another place on the same displayed image. (2) In a computer program, a conditional instruction and an indicator to be interrogated by that instruction. (3) A device or programming technique for making a selection, for example, a toggle, a conditional jump.

switch list - See *Task List* .

symbolic identifier - A text string that equates to an integer value in an include file, which is used to identify a programming object.

symbols - In Information Presentation Facility, a document element used to produce characters that cannot be entered from the keyboard.

synchronous - Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals. (T) See also *asynchronous* .

System Menu - In the Presentation Manager, the pull-down in the top left corner of a window that allows it to be moved and sized with the keyboard.

System Object Model (SOM) - A mechanism for language-neutral, object-oriented programming in the OS/2 environment.

system queue - The master queue for all pointer device or keyboard events.

system-defined messages - Messages that control the operations of applications and provides input an other information for applications to process.

Systems Application Architecture (SAA) - A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

Glossary - T

table tags - In Information Presentation Facility, a document element that formats text in an arrangement of rows and columns.

tag - (1) One or more characters attached to a set of data that contain information about the set, including its identification. (I) (A) (2) In Generalized Markup Language markup, a name for a type of document or document element that is entered in the source document to identify it.

target object - An object to which the user is transferring information.

Task List - In the Presentation Manager, the list of programs that are active. The list can be used to switch to a program and to stop programs.

terminate-and-stay-resident (TSR) - Pertaining to an application that modifies an operating system interrupt vector to point to its own location (known as hooking an interrupt).

text - Characters or symbols.

text cursor - A symbol displayed in an entry field that indicates where typed input will appear.

text window - Also known as the VIO window.

text-windowed application - The environment in which the operating system performs advanced-video input and output operations.

thread - A unit of execution within a process. It uses the resources of the process.

thumb mark - The portion of the scroll bar that describes the range and properties of the data that is currently visible in a window. Also known as a *slider box* .

thunk - Term used to describe the process of address conversion, stack and structure realignment, etc., necessary when passing control between 16-bit and 32-bit modules.

tilde - A mark used to denote the character that is to be used as a mnemonic when selecting text items within a menu.

time slice - (1) An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing-unit time is allocated to another task, so a task cannot monopolize processing-unit time beyond a fixed limit. (2) In systems with time sharing, a segment of time allocated to a terminal job.

time-critical process - A process that must be performed within a specified time after an event has occurred.

timer - A facility provided under the Presentation Manager, whereby Presentation Manager will dispatch a message of class WM_TIMER to a particular window at specified intervals. This capability may be used by an application to perform a specific processing task at predetermined intervals, without the necessity for the application to explicitly keep track of the passage of time.

timer tick - See *clock tick* .

title bar - In SAA Advanced Common User Access architecture, the area at the top of each window that contains the window title and system menu icon. When appropriate, it also contains the minimize, maximize, and restore icons. Contrast with *panel title* .

TLB - Translation lookaside buffer.

transaction - An exchange between a workstation and another device that accomplishes a particular action or result.

transform - (1) The action of modifying a picture by scaling, shearing, reflecting, rotating, or translating. (2) The object that performs or defines such a modification; also referred to as a *transformation* .

Translation lookaside buffer (TLB) - A hardware-based address caching mechanism for paging information.

Tree - In the Presentation Manager, the window in the *File Manager* that shows the organization of drives and directories.

truncate - (1) To terminate a computational process in accordance with some rule (A) (2) To remove the beginning or ending elements of a string. (3) To drop data that cannot be printed or displayed in the line width specified or available. (4) To shorten a field or statement to a specified length.

TSR - Terminate-and-stay-resident.

Glossary - U

unnamed pipe - A circular buffer, created in memory, used by related processes to communicate with one another. Contrast with *named* pipe.

unordered list - In Information Presentation Facility, a vertical arrangement of items in a list, with each item in the list preceded by a special character or bullet.

update region - A system-provided area of dynamic storage containing one or more (not necessarily contiguous) rectangular areas of a window that are visually invalid or incorrect, and therefore are in need of repainting.

user interface - Hardware, software, or both that allows a user to interact with and perform operations on a system, program, or device.

User Shell - A component of OS/2 that uses a graphics-based, windowed interface to allow the user to manage applications and files installed and running under OS/2.

utility program - (1) A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program. (T) (2) A program designed to perform an everyday task such as copying data from one storage device to another. (A)

Glossary - V

value set control - A visual component that enables a user to select one choice from a group of mutually exclusive choices.

vector font - A set of symbols, each of which is created as a series of lines and curves. Synonymous with *outline font* . Contrast with *image font* .

VGA - Video graphics array.

view - A way of looking at an object's information.

viewing pipeline - The series of transformations applied to a graphic object to map the object to the device on which it is to be presented.

viewing window - A clipping boundary that defines the visible part of model space.

VIO - Video Input/Output.

virtual memory (VM) - Synonymous with *virtual storage* .

virtual storage - (1) The storage space that may be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of auxiliary storage available, not by the actual number of main storage locations. (I) (A) (2) Addressable space that is apparent to the user as the processor storage space, from which the instructions and the data are mapped into the processor storage locations. (3) Synonymous with *virtual memory* .

visible region - A window's presentation space, clipped to the boundary of the window and the boundaries of any overlying window.

volume - (1) A file-system driver that uses a block device driver for input and output operations to a local or remote device. (I) (2) A portion of data, together with its data carrier, that can be handled conveniently as a unit.

Glossary - W

wildcard character - Synonymous with *global file-name character* .

window - (1) A portion of a display surface in which display images pertaining to a particular application can be presented. Different applications can be displayed simultaneously in different windows. (A) (2) An area of the screen with visible boundaries within which information is displayed. A window can be smaller than or the same size as the screen. Windows can appear to overlap on the screen.

window class - The grouping of windows whose processing needs conform to the services provided by one window procedure.

window coordinates - A set of coordinates by which a window position or size is defined; measured in device units, or *pels* .

window handle - Unique identifier of a window, generated by Presentation Manager when the window is created, and used by applications to direct messages to the window.

window procedure - Code that is activated in response to a message. The procedure controls the appearance and behavior of its associated windows.

window rectangle - The means by which the size and position of a window is described in relation to the desktop window.

window resource - A read-only data segment stored in the .EXE file of an application o the .DLL file of a dynamic link library.

window style - The set of properties that influence how events related to a particular window will be processed.

window title - In SAA Advanced Common User Access architecture, the area in the title bar that contains the name of the application and the OS/2 operating system file name, if applicable.

Workplace Shell - The OS/2 object-oriented, graphical user interface.

workstation - (1) A display screen together with attachments such as a keyboard, a local copy device, or a tablet. (2) (D of C) One or more programmable or nonprogrammable devices that allow a user to do work.

world coordinates - A device-independent Cartesian coordinate system used by the application program for specifying graphical input and output. (I) (A)

world-coordinate space - Coordinate space in which graphics are defined before transformations are applied.

WYSIWYG - What-You-See-Is-What-You-Get. A capability of a text editor to continually display pages exactly as they will be printed.

Glossary - X

There are no glossary terms for this starting letter.

Glossary - Y

There are no glossary terms for this starting letter.

Glossary - Z

z-order - The order in which sibling windows are presented. The topmost sibling window obscures any portion of the siblings that it overlaps; the same effect occurs down through the order of lower sibling windows.

zooming - The progressive scaling of an entire display image in order to give the visual impression of movement of all or part of a display group toward or away from an observer. (I) (A)

8.3 file-name format - A file-naming convention in which file names are limited to eight characters before and three characters after a single dot. Usually pronounced "eight-dot-three." See also *non-8.3 file-name format* .

Notices

November 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "(C) (your company name) (year). All rights reserved."

(C) Copyright International Business Machines Corporation 1994,1996. All rights reserved.
Note to U.S. Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:
AT
Common User Access
CUA
IBM
IBM PC AT
OS/2
Personal System/2
Presentation Manager
PS/2
SAA
Systems Application Architecture

The following terms are trademarks of other companies:

Intel	Intel Corporation
